Jackson Cunningham Woodruff

An Optimising Compiler for ML

Computer Science Tripos: Part II
Magdalene College
May 2018

Proforma

Name:	Jackson Cunningham Woodruff
College:	Magdalene College
Project Title:	An Optimising Compiler for ML
Examination:	Computer Science Tripos - Part II, 2018
Word Count:	11173
Project Originator:	Jackson Cunningham Woodruff
Supervisor:	Dr Timothy Jones

Original Aims of the Project

The original aim of this project was to write an optimising compiler from SML source code to Java bytecode. Various optimisations were to be explored and compared in their effectiveness. The compiler was to be compared to existing ML compilers. This dissertation explains the design process, the design decisions made, and evaluates the final product.

Work Completed

I have implemented an optimising compiler from SML source to Java bytecode. Optimisations implemented are: a peephole optimiser, a tree simplifier, dead store elimination, copy propagation, constant propagation, function inlining and tail-call elimination. A testsuite has been written and maintained. A benchmarking suite has been constructed, and benchmarking scripts written. The optimisations have been compared and their improvements documented. Success criteria were met.

Special Difficulties

None.

Declaration of Originality

I, Jackson Cunningham Woodruff of Magdalene College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Jackson Woodruff

Date

May 12, 2018

Contents

1 Introduction **4**

2 Preparation **6**
- 2.1 Source Language . . . 6
- 2.2 Target Language . . . 6
- 2.3 Selecting Optimisations . . . 7
- 2.4 Benchmarking . . . 8
- 2.5 Compilation Pipeline . . . 8
- 2.6 User Interaction . . . 9
- 2.7 Development Strategy . . . 10
- 2.8 Testability . . . 10
- 2.9 Selecting Tools . . . 10
- 2.10 Starting Point . . . 11

3 Implementation **12**
- 3.1 Compiler Implementation Details . . . 15
  - 3.1.1 Verification . . . 15
  - 3.1.2 Designing TIR . . . 16
  - 3.1.3 Datatype Lifting . . . 17
  - 3.1.4 Outlining . . . 17
  - 3.1.5 Lambda Lifting . . . 19
  - 3.1.6 Let Elimination . . . 20
  - 3.1.7 Pattern Match Elimination . . . 22
  - 3.1.8 Variable Numbering . . . 23
- 3.2 Representing TIR with Bytecode . . . 23
- 3.3 Optimisation Implementation Details . . . 27
  - 3.3.1 Tail-Call Elimination Pass . . . 27
  - 3.3.2 Function Inlining . . . 28
  - 3.3.3 Copy Propagation Pass . . . 29
  - 3.3.4 Simplify Passes . . . 29
  - 3.3.5 Dead Store Elimination Pass . . . 31
  - 3.3.6 Peephole Pass . . . 31
  - 3.3.7 Phase Ordering Problem . . . 33
- 3.4 Testsuite Implementation Details . . . 36
- 3.5 Benchmarking Implementation Details . . . 37
- 3.6 Standard Library Implementation Details . . . 38

3.7 Compilation Scripts Implementation Details 38
3.8 Summary 38

4 Evaluation **40**
4.1 Methodology 40
4.2 Execution Time 40
4.3 Compile Time 49
4.3.1 Typechecker 49
4.4 Code Size 56
4.5 Additional Optimisations 59
4.6 Summary 59

5 Conclusion **60**
5.1 Further Work 60
5.2 Summary 61

Appendices **66**

A Inlining Performance Profiling in Java **67**

B Scala Garbage Collector **70**

Chapter 1

Introduction

Compilers are a crucial tool in all software projects. Compilers must be correct, and companies' fortunes can depend on the speed of generated code. Free compilers are critical for the success of small companies. My project aims to provide a free SML compiler as another possible tool.

There are existing SML compilers, notably SML/NJ, MosML and PolyML. These compilers all provide interactive sessions and compilation facilities. However, the compilation targets are non-portable machine code or obscure virtual machines. These projects have too few contributors to maintain backends or virtual machines for many targets. For example, SML/NJ does not support Arm as a compilation target [1]. Many of these compilers do not produce machine-independent executables, and so do not benefit from the "compile once, run everywhere" philosophy of projects like the JVM. Further, the nature of native executables means that they do not benefit from a JIT making dynamic optimisation decisions. A similar issue exists in that smaller virtual machines do not have the same range of optimisations as the JVM. Speed of produced executables is important for ML programs. For example, the HOL project, one of the largest active SML projects, moved from MosML to PolyML largely due to speed issues as discussed by Norrish [2]. Even with a JIT, compile-time optimisations can be important, as discovered by the SOOT project [3].

Finally, each compiler has small incompatibilities with others in the implementation of the standard libraries. A program written for MosML may not be compatible with PolyML. If one project stops supporting new architectures — a situation that poses significant development costs, the task of migrating a program written for one compiler to another may be non-trivial. A program written for my compiler (MLC) will require that my project receives bug fixes, but not that it adds new targets.

My project is aimed at the JVM. The JVM is a larger project than any of the ML compilers, and it supports a vast range of targets. More targets have support through alternate JVMs. The executables generated are machine-independent and are optimised at runtime by the JIT with up-to-date techniques on every run. For ML programs to take advantage of this, we needed ML-to-Java bytecode compiler.

As discussed by Hughes [4], functional languages have benefits for certain

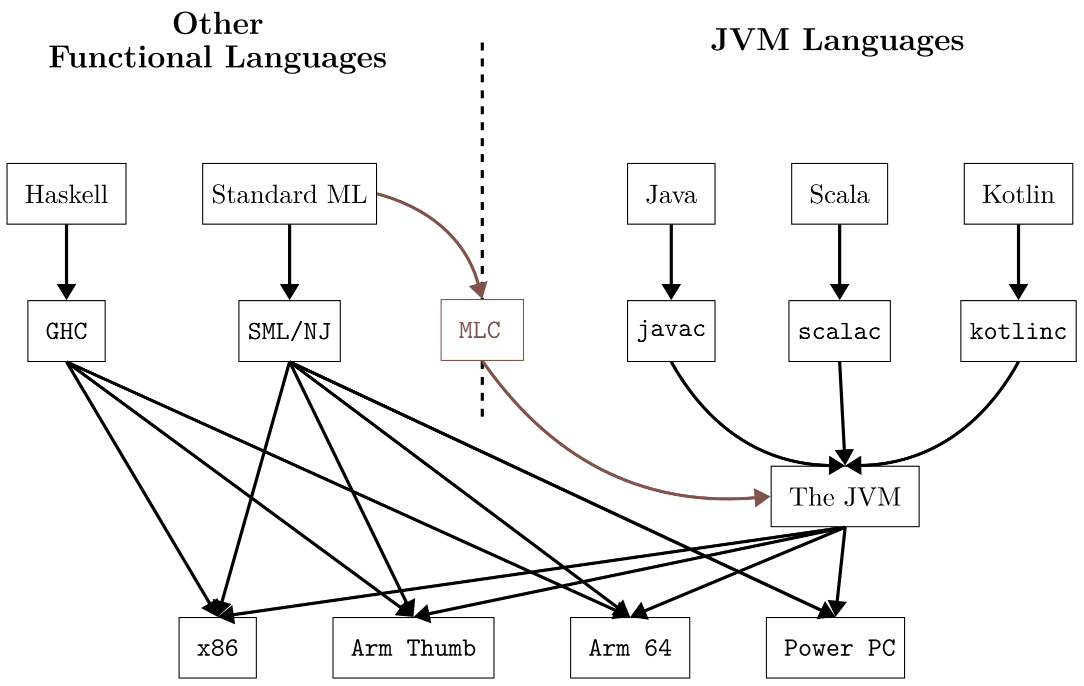


Figure 1.1: How my project (MLC) fits into current compiler infrastructure for the JVM.

---

tasks. For example, they eliminate major sources of bugs. However, Java's prevalence in industry makes functional languages difficult to integrate with existing projects. A compiler targeting the JVM can easily interoperate with Java; this is considered a key feature for many JVM languages (e.g. Scala [5]). Figure 1.1 shows how this project fits.

An SML-to-Java bytecode compiler already exists: MLj [6]. However, it is no longer maintained. I was unable to compile MLj with my current version of Ubuntu and SML/NJ. Further, MLj does not implement tail-call elimination, which unnecessarily restricts the programmer to limited JVM stack size.

More competition is good for the free compiler market. For instance, as a direct result of market pressure from LLVM [7], GCC has adopted useful features such as better quality warnings. Likewise, a complete version of this project would drive support from existing compilers for features that are core to a complete version of this project (such as interoperability with Java).

My project implements a compiler that produces JVM bytecode from SML source to resolve these issues. As a result of time constraints, the project is not a complete language compiler and implements only a pure subset of SML. However, my project has been designed with the entire language in mind. The extensions enabling this can build on what has been implemented.

Chapter 2

Preparation

In this chapter, I discuss the selection of Standard ML as defined in "The Definition of Standard ML" [8] as an input language, JVM bytecode as a target language, and Scala as an implementation language. In addition, I discuss decisions about tracking performance and testing MLC. Finally, the selection of optimisations is discussed.

2.1 Source Language

Haskell was my original choice of language due to its popularity. However, I felt that SML provided a better starting point due to my prior knowledge. Further, the pure subset of SML I intend to compile is the same as that subset of Haskell (and many other functional languages with pure subsets, such as OCaml). The compromise was to plan the mid-end in a language-agnostic manner that easily extends to languages such as Haskell.

SML is too large a language to compile for this project. I selected a pure subset described in Figure 3.1 that provides expressivity without bloating any passes of my compiler (e.g., lots of syntactic sugar would complicate the parser in comparison to other stages).

Further, I intended for it to be possible to eventually add all omitted features (from the core language defined in the definition [8]). Crucially, the compilation techniques used must play well with omitted features. The most notable omitted feature is references. The implementation has shown that references would be a simple extension.

2.2 Target Language

Selecting the JVM was not particularly difficult given the limited options. Targeting a particular architecture would be difficult due to time constraints and the additional complexities of processes such as register allocation and conforming with ABIs. I considered LLVM IR, but this does not have the desirable

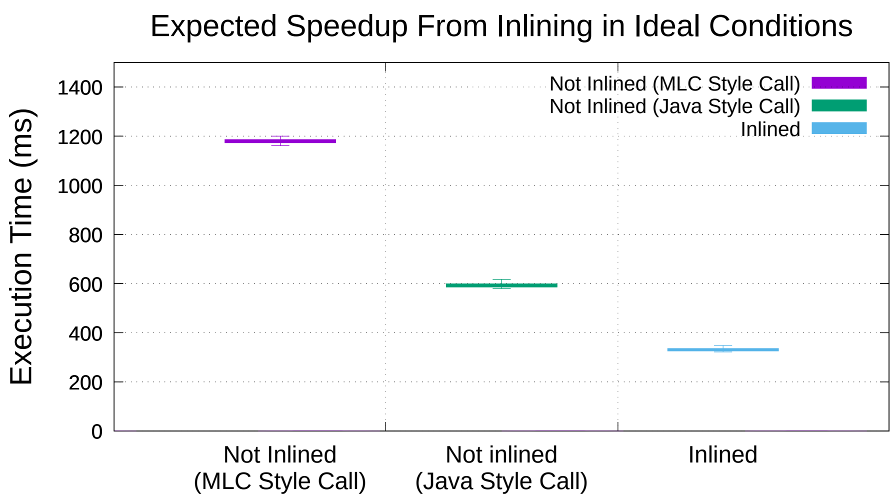


Figure 2.1: Estimation of possible performance improvements for inlining on the JVM. Method calls are made in a tight loop. Code for this is given in Appendix A. Performance improvements of this magnitude are not expected in real programs.

"single executable for all platforms" traits and is more complicated. Further, LLVM has fewer supported architectures than Java bytecode.

I considered other JITed instruction formats such as Python's `pyc` format or Microsoft's intermediate language `MISL`. However, none appeared to be as documented or supported as Java bytecode. Further, many other languages (notably Scala, Groovy and Kotlin) have used the JVM as a compilation target; so, it is known to be feasible.

A further consideration in target selection is legal feasibility. The Java bytecode specification is licensed under the Limited License Grant [9]. This provides the freedom for two critical aspects of this project: first, clause 1(i) allows applications to be developed targeting an implementation of the specification. Second, clause 1(ii) allows discussion of the specification with third parties;[1] this clause is important for this document.

2.3 Selecting Optimisations

Two criteria were used to select optimisations: the first was to find optimisations with expected impact on generated code speed. The second was to identify the optimisations needed to clean up after previous passes.

Previous research has shown compile-time optimisations such as loop unrolling, loop unfolding and loop invariant code motion can provide large perfor-

[1]Provided large sections are not copied verbatim.

mance benefits [10]. SOOT (a Java bytecode optimiser) has recorded function inlining yielding an 8% performance improvement [3]. I attempted to verify this as this study was undertaken in 2002. As shown in Figure 2.1, I was able to reproduce this performance increase with traditional Java methods. Using functions in a similar style to my representation (discussed in Section 3.2), I found significant performance gains. The inlining was performed manually in the source, and compared to code without inlining.

With these data, I identified that function inlining, partial inlining of recursive functions and tail-call optimisation were likely to be particularly profitable.

The selection of optimisations to clean up after compilation passes posed a significant challenge. It proved difficult to predict which issues each compilation stage would encounter. I approached this problem by selecting well-known optimisations. For the most part, this worked well. Optimisation passes remove most junk code generated by MLC.

2.4 Benchmarking

Performance tracking was detailed from the outset. Making informed decisions about which optimisations are helpful and which are not requires data on their effects. Some data can only be collected after an optimisation has been implemented. These data have proven useful for inspecting why optimisations did or did not work in certain cases.

I planned to track benchmark execution time and executable size. The Linux utility `perf` [11] yields counts of specific events in the processor. I intended for this to reveal detailed effects of optimisations on the microarchitecture.

My benchmarks were to be designed so that all represented different tasks and performed real algorithms. It makes little practical sense to implement a compiler that optimises only toy cases. The tasks aimed to match those in current benchmarking suites. However, there were some restrictions. Critically, I had to ensure that the benchmarks would run on a system with limited stack depth and without tail-call elimination. I was careful to consider common benchmarking traps described by SPEC [12]. From a quantitative perspective, these are hard to avoid; however, my benchmarks have been designed with these pitfalls in mind. I believe my benchmarks avoid them.

The selection of benchmarks aided other areas of planning. Because MLC had not been written when I wrote the benchmarks, the selection of the benchmarks proved useful in picking my language subset. Writing benchmarks that perform actual tasks was sufficient to ensure expressiveness of my subset.

2.5 Compilation Pipeline

I proposed three intermediate structures: The first, called AST (Abstract Syntax Tree), is very similar to the structure of SML. The second, called TIR (Tree Intermediate Representation), was designed to be a general structure to represent any functional language with minor additions but maintain similarity to SML. The final representation, called ByteR (Bytecode Representation), bears

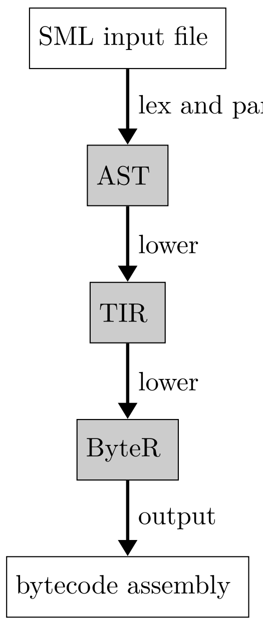


Figure 2.2: Schematic for the pipeline of intermediate languages in MLC (mine shaded in grey).

---

an instruction for instruction correspondence to JVM bytecode. The layout of intermediate languages is shown in Figure 2.2.

Before writing code, I had difficulty providing more detail about representation structure. Instead, I specified the tasks each structure should facilitate. I wanted clean separation between compiler stages: the source-language-dependent features, the optimisations, and the target-language-dependent features. To achieve this, I settled on three main intermediate languages:

AST: The first stage designed for language dependent tasks such as typechecking.

TIR: The second stage designed to model a generic functional language. This is where I imagined the optimisations would take place.

ByteR: The final stage to map to the target language cleanly.

This layout was chosen so MLC is not limited to a single frontend or single target.

2.6 User Interaction

I intended to provide a flexible and intuitive interface as an access point for the user. A command line executable fits this criterion for a compiler.

Users expect to interact with a compiler via the command line. A compiler must be usable in scripts, where a command line interface is critical. Further, a command line executable can be integrated into an IDE. I intended for command line flags to control MLC.

2.7 Development Strategy

In my proposal, I stated I would take a waterfall approach to development. I intended to use one for each compilation pass. In practice, this worked well for some passes (such as the lowering passes) where a requirements document suited the problem at hand. In other cases where I found it difficult to fully anticipate all potential issues, I adopted an iterative approach.

2.8 Testability

MLC has the benefit of being deterministic and having machine-readable input and output. For testing, it can be treated as a black box. Mutable state can make unit testing difficult, particularly if this mutability is subtle. Running a new instance of the compiler for every test provides a solution. The optimum is to have the compiler output information that can be checked automatically for something expected.

After writing each pass, I added the option to produce a dump file. Each dump file displays in text format the current intermediate representation and any additional information relevant to that pass (e.g. reasoning behind inlining decisions). This allows tests verifying the entire compilation pipeline up to a new pass.

I intended for the testsuite to ensure regressions will be tracked and added features will remain enabled. By testing after each pass, issues can be caught where they occur. The testsuite was designed to be compiler agnostic. See Section 3.4 for more detail.

I also required that scripts will interact cleanly with the test interface. The command line interface described in Section 2.6 can accept flags for outputting various intermediate forms. By ensuring each pass can output the intermediate form after running, tests check individual passes for correctness in addition to the compiler as a whole. To attempt to satisfy these requirements, I proposed the development of a testbench that runs a set of SML files through the compiler.

I settled on the design principle of ensuring MLC never fails silently. Further, when an error is encountered, it should occur as early as possible. This is to be achieved with assertions and specialised verification passes that ensure tricky transformations have been implemented correctly.

2.9 Selecting Tools

I chose Scala as an implementation language because it meets the “ease of implementation” requirements. My rationale for choosing Scala is further justified in the proposal. For a build tool, I selected Simple Build Tool (SBT). SBT provides library management and partial rebuilds among other features, but it is a notoriously difficult tool with which to work. Nevertheless, no other options provide integration with crucial build-tool plugins for my development process (such as one-jar [13], which allows MLC to be packaged into a single executable jar file).

Version control is non-negotiable. It provides a roll-back option should something go wrong. It helps with backup and keeping code synchronised between machines. I picked Git due to previous experience.

Documentation is recorded on several levels. Each pass contains a `README.md` file that specifies the behaviour of that pass. Each non-trivial file contains a block comment explaining its purpose. Each non-trivial method has a block comment describing its parameters and behaviour. Line comments are used in particularly difficult code. A new contributor should start by reading the `README.md` files. A new user should start by reading the `--help` option.

My project is on Github [14]. Keeping it online meets my proposal's backup criteria[2] and makes MLC easy to distribute. Making MLC free is good for the long-term health of the project.

Continuous integration is a useful tool. It has prevented issues from building up and ensures the online version works. For this project, I use TravisCI [15] due to previous experience.

Python is a flexible language popular for scripting tasks. It fits the requirements for testsuite and benchmarking scripts. Those are, no speed requirements and a need for sensible access to the command line.

2.10 Starting Point

In addition to Scala standard libraries, Scala provides a parser generator, the `ParserCombinators` library [16] which I used to construct my grammar. For command line argument processing, I use an SBT plugin `scallop` [17]. The bytecode assembler, Krakatau, is on Github [18].

In my Python scripts, I use the Python libraries. I made minor adaptations to one benchmark from Cumming [19] and a host of benchmarks from the MLton benchmarking suite (described in more detail in Section 3.5). One testsuite folder (`test/general/`) is dedicated to tests from online ML tutorials. Each test cites its source and licence if specified. My changes to these tests are minor, clearly marked and compatible with the licence.

For tracking performance during the project, I used the LLVM Nightly Testing [20] framework. For drawing graphs, I use GNUPlot [21] and matplotlib [22].

[2]In addition, I keep snapshots from the last 30 days in a Dropbox folder. All major versions are backed up with the CL backup on my benchmarking machine.

Chapter 3

Implementation

I have implemented a compiler that first lexes and parses SML source. This is followed by typechecking and a series of lowering passes that eventually produce Java bytecode. I have implemented bare-bones standard libraries as specified in my proposal. These are written in Java and compatible with my generated code. Finally, MLC contains an extensive testsuite of SML files and a benchmarking suite. Both have their own run scripts written in a compiler independent manner.[1]

MLC verifies SML code for syntax and type correctness, and implements the transformation from SML source into the JVM bytecode assembly format specified by Krakatau. Krakatau is a GPLv2-licensed JVM bytecode assembler. The assembler outputs a number of Java `.class` files. These can be passed to the `jar` utility [23] with the standard library `.class` files to create a runnable `.jar` file.

The grammar I have implemented is shown in Figure 3.1.

My project is licensed with the GNU GPLv3. This ensures compatibility with Krakatau's licence should it become desirable to use Jython (an interface between Java and Python) to avoid the Python dependency for the compiler.[2]

I have also written five benchmarks. To augment these benchmarks, I have gathered suitable[3] benchmarks from other suites. One has been adapted from an ML tutorial [19]. I have adapted an additional four benchmarks for MLC from MLton's benchmark suite [26]. These were adapted in the last phase of the project, and so they were not tracked over time. The measurement infrastructure in the benchmarking suite is adaptable to any compiler.

I have compiled a large and varied testsuite for my subset of SML. The testsuite infrastructure is also adaptable to any compiler.

[1]Because scripts were written before the creation of the compiler they were tested with MosML.

[2]Currently this is not necessary as the assembler is treated as a separate program by the shell script. Using Jython would make this a borderline case, as discussed in the GNU GPL FAQ [25].

[3] I used benchmarks that required few changes to fit my implemented subset.

Constants:	*con*	::=	*float* | *int* | *char* | *string* | `true` | `false` | `()` | `[]`
Identifiers:	*id*	::=	`[A-z_][A-z0-9_‘]`*
	infix_id	::=	`[+-*/]` | `^` | `div` | `mod` | `<>` | `<` | `>` | `<=` | `>=` | `::` | `@`
	unary_id	::=	`~` | `not` | `print`
	tyvar	::=	`‘[A-z‘_]`* | `‘‘[A-z‘_]`*
	tycon	::=	`bool` | `char` | `exn` | `int` | `list` | `real` | `string` | `unit`
	longid	::=	*id*. ··· .*id*
Types:	*typ*	::=	*tyvar* | *id* | (*typ*) | *typ* `->` *typ* | *typ* ∗ ··· ∗ *typ*
Expressions:	*exp*	::=	*con*
			longid
			exp exp
			exp; {*exp*}
			exp infix_id exp
			unary_id exp
			(*exp*, ..., *exp*)
			[*exp*, ..., *exp*]
			`let` *dec* `in` *exp* `end`
			exp : *typ*
			`if` *exp* `then` *exp* `else` *exp*
			exp `andalso` *exp*
			exp `orelse` *exp*
			`case` *exp* `of` *match*
			`fn` *match*
			exp `handle` *match*
			`raise` *exp*
	match	::=	*pat* `=>` *exp* {‘|’ *match*}
Patterns:	*pat*	::=	_
			id
			con
			pat :: *pat*
			(*pat*, ..., *pat*)
			[*pat*, ..., *pat*]
			pat : *typ*
	valpat	::=	_ | *id* | (*valpat*, ..., *valpat*)
Declarations:	*dec*	::=	`val` *valbind*
			`fun` *funbind*
			`exception` *datconstr*
			`datatype` *datbind*
			dec{;} *dec*
	datbind	::=	*id* `=` *datconstr* ‘|’ ··· ‘|’ *datconstr*
	datconstr	::=	*id* `of` *typ*
	valbind	::=	*valpat* `=` *exp*
	funbind	::=	*id pat* ··· *pat* {: *typ*} `=` *exp* {‘|’ *funbind*}

Figure 3.1: The grammar I have implemented. Details of individual lexemes omitted. ‘|’ is used to represent the input string | and | is used to mean “or”. Elements between {} are optional. Comments, which occur between `(*` and `*)` are valid at all locations where whitespace is valid. Adapted from the grammar by Rossberg with derived forms [24].

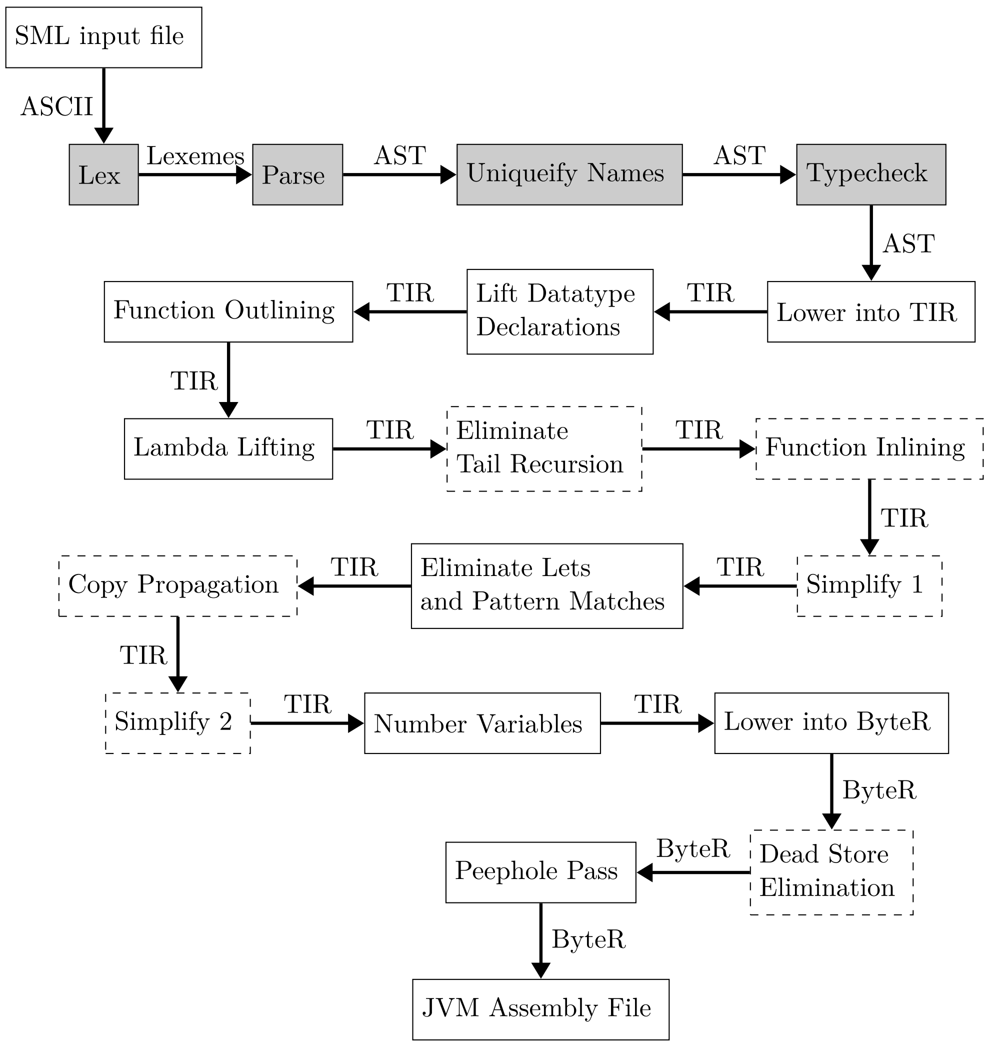


Figure 3.2: Schematic for the pipeline of intermediate languages. Optional optimisation passes are in dashed boxes. Verification passes shaded in grey.

The top-level design of the compiler is an ordered list of passes. The ordering of passes is naturally sequential.[4] The output of one pass is provided as input to the next pass. Optional passes act as no-ops when disabled.

3.1 Compiler Implementation Details

The full pipeline of the compiler is shown in Figure 3.2. There are two main stages in the compiler: the first stage verifies the validity of input SML code and transforms the source into the AST representation. The second stage takes the TIR representation, which is initially similar to the AST representation, and transforms it so that it can easily be represented in bytecode.

3.1.1 Verification

The verification stage can itself be broken down. The first step involves lexing and parsing. These are implemented with the `ParserCombinators` library.

The lexer and parser detect syntactic errors in inputs. If none are found, the parsed program is passed to the name uniquifier.

The name uniquifier takes as input an AST representation and changes the names of the variables. After this, each variable name identifies, at most, one variable. Name reuse is problematic for later stages that depend on information derived by typechecking. The typechecker derives all types; however, without differences between occurrences of same variable name `x`, the type of one occurrence may be lost.

```
val x = 10
val x = "Value is " ^ Int.toString(x)
```

Figure 3.3: An example where new names are required.

Consider Figure 3.3. It is important to know the type of the first `x` is `int` at the invocation of `Int.toString` so it can be correctly cast. However, with a single environment, the value of `env(x)` can only be a single result.

The cleanest solution is to distinguish between the first and the second `x`, which I achieve by making each name unique. A similar problem arises in lambda lifted functions.

The typechecker is the most substantial part of the verifier. I have implemented Hindley-Milner using algorithm-W as described by Damas and Milner [27]. A substantial portion of time spent in the typechecker is spent in the unifier and type environment.

[4]In order to achieve good multicore performance, multiple instances of a compiler may be invoked on different source files.

Environment Representation

I chose to implement the type environment as a map from variable name to type. This is significantly more efficient than some alternatives (e.g. a linked list). However, mutable maps do not implicitly deal with nesting issues the way linked lists do.

As a result, the type environment structure is stored as a hierarchy of maps, one map for each level of nesting. Lookups propagate through this hierarchy. This approach is not strictly necessary when names are unique. However, it is faster to apply operations that walk over every type in a particular nesting of an environment if nestings do not contain every variable.

Unifiers are maps from type variable to type. Upon application of a unifier to a type environment, the entire environment must be walked. Type variables mapped by the unifier are looked up and changed in the environment. The unifier ensures unifications are complete. For example, if a unifier maps $\alpha \to \beta, \beta \to \gamma$ then α must map to γ. This is performed by tracing each type variable through the unifier. Cyclical type reductions are possible. The unifier must keep track of types it has already seen when unifying a cyclic chain. If there is some chain $\alpha \to \beta \to \gamma \to \alpha$, the unifier must ensure each of these type variables map to the same result type variable (say α).

Unifiers have iterative components to them. Most of the compiler recursively walks the tree and so any "infinite" loop results in an overflow. The unifier is implemented using a while loop, which could fail by looping endlessly. This has been resolved here, as elsewhere, by capping the number of iterations to a large value (significantly larger than viable in practice due to limitations elsewhere). If it is exceeded, an exception is thrown.

Typechecker Bugs

My typechecker has the only known bugs in my project. This is based on the faulty assumption a type variable must be forall scoped when it is used at a function call site. This is not necessarily the case as in Figure 3.4. This is currently accepted, but should be rejected.

This is difficult to address because it requires the addition of a tag on every type variable. I estimate that this would take another week to implement.

3.1.2 Designing TIR

TIR is my optimisation structure. It also acts as a translation language that is lowered progressively. To achieve this, TIR has an ML-like subset and a subset that introduces features much closer to the underlying bytecode.

There are three main forms of TIR:

1. Before lambda lifting: similar to SML structure.

2. After lambda lifting: all functions are lambda lifted to the top level. Closures created where necessary.

```
fun f x =
  let
    fun g y = f x
  in
    g
  end
```

Figure 3.4: In this example, the use of `f` should not be forall qualified because it is within the declaration of `f`. Instead, my typechecker sees `f` to have type $\forall\alpha,\beta.\alpha\rightarrow\beta$ and gives `g` the type $\gamma\rightarrow\delta$ (because the use of `f` is instantiated with fresh variables since). This should be rejected as it should introduce a cyclical type.

3. After pattern match elimination: all functions contain a single case. Nested `let` declarations are eliminated. Case statements are compiled into `if` statements.

With more time, these structural requirements would ideally be enforced using more intermediate languages. Instead, these structural requirements are verified by verification passes. This was done to reduce implementation load. The other value of using one main intermediate language while constructing the compiler is flexibility. New node types may be added as needed (which is regularly in a rapidly evolving compiler) without the cost of creating a new intermediate language. A single optimisation language also means analysis passes need not be duplicated.

Further, rebuilding the entire program tree on each pass is wasteful. There are garbage collection problems for compilers with immutable trees. See Appendix B for details. TIR nodes are mutable to deal with these issues. This approach also offers more flexibility. Nodes may have values assigned in callbacks rather than requiring assignments at the point of encounter during a walk.

The semantics of the nodes in the TIR are described in Figure 3.5.

3.1.3 Datatype Lifting

Datatype lifting takes datatypes not declared at the top level and moves them to the top level. It is similar to lambda lifting. Because there are fewer complexities (datatypes do not need closure analysis) this is performed earlier in the pipeline. The functionality is very similar to that in Section 3.1.5. An example is shown in Figure 3.6.

3.1.4 Outlining

Outlining is a space optimisation in many compilers (e.g. LLVM [28]). My outliner is designed without an optimisation cost model (although the application of the transformation is identical). The need for this pass is best described through an example.

Static Semantics:

$$\text{TExpWhileTrue}\ \frac{E \vdash \mathit{exp} \Rightarrow \tau\,\texttt{loop}}{E \vdash \texttt{while (true)}\ \mathit{exp} \Rightarrow \tau}$$

$$\text{TExpContinue}\ \frac{}{E \vdash \texttt{Continue} \Rightarrow \tau\,\texttt{loop}}$$

$$\text{TExpBreak}\ \frac{E \vdash \mathit{exp} \Rightarrow \tau}{E \vdash \texttt{Break}(\mathit{exp}) \Rightarrow \tau\,\texttt{loop}}$$

$$\text{TExpFunLet}\ \frac{E \oplus \{v_1 \rightarrow \alpha_1, \ldots, v_n \rightarrow \alpha_n\} \vdash \mathit{exp} \Rightarrow \tau \qquad \text{Where } \alpha_1, \ldots, \alpha_n \text{ are the types of the expressions assigned to } v_1, \ldots, v_n}{E \vdash \texttt{funlet}\, v_1, \ldots, v_n\, \texttt{in}\, \mathit{exp}\, \texttt{end} \Rightarrow \tau}$$

$$\text{TExpListExtract}\ \frac{E \vdash \mathit{exp} \Rightarrow \tau\ \mathit{list}}{E \vdash \mathit{exp}[n] \Rightarrow \tau}$$

$$\text{TExpAssign}\ \frac{E \vdash \mathit{exp} \Rightarrow \tau \qquad v \in E \qquad E(v) = \tau}{E \vdash \texttt{assign}\, v\, \texttt{to}\, \mathit{exp} \Rightarrow \mathit{unit}}$$

Dynamic Semantics:

$$\text{TExpWhileTrue (1)}\ \frac{s, E \vdash \mathit{exp} \Rightarrow s', \texttt{Continue} \qquad s', E \vdash \texttt{while (true)}\ \mathit{exp} \Rightarrow s'', v}{s, E \vdash \texttt{while (true)}\ \mathit{exp} \Rightarrow s'', v}$$

$$\text{TExpWhileTrue (2)}\ \frac{s, E \vdash \mathit{exp} \Rightarrow s', \texttt{Break}(\mathit{exp}') \qquad s', E \vdash \mathit{exp}' \Rightarrow s'', v}{s, E \vdash \texttt{while (true)}\ \mathit{exp} \Rightarrow s'', v}$$

$$\text{TExpFunLet}\ \frac{s, E \vdash \mathit{exp} \Rightarrow s', v}{s, E \vdash \texttt{funlet}\, v_1, \ldots, v_n\, \texttt{in}\, \mathit{exp}\, \texttt{end} \Rightarrow s', v}$$

$$\text{TExpListExtract (1)}\ \frac{s, E \vdash \mathit{exp} \Rightarrow s', l \qquad \mathit{length}(\mathit{exp}) > n}{s, E \vdash \mathit{exp}[n] \Rightarrow s', \mathit{hd}(\underbrace{\mathit{tl}(\cdots \mathit{tl}(l) \cdots)}_{n \text{ times}})}$$

$$\text{TExpListExtract (2)}\ \frac{s, E \vdash \mathit{exp} \Rightarrow s', l \qquad \mathit{length}(\mathit{exp}) \leq n}{s, E \vdash \mathit{exp}[n] \Rightarrow s', \text{Runtime Exception}}$$

$$\text{TExpAssign}\ \frac{s, E \vdash \mathit{exp} \Rightarrow s', x}{s, E \vdash \texttt{assign}\, v\, \texttt{to}\, \mathit{exp} \Rightarrow s' + \{v \rightarrow x\}, ()}$$

Figure 3.5: The static and dynamic semantics of TIR. `Continue` and `Break` are internal monads of type τ `loop`. This should be interpreted in addition to the semantics of SML [8] as all SML terms retain their semantics.

```
val _ =
  let
    datatype tree =
        Leaf of int
      | Node of tree * tree
    exception Fail of int
  in
    ...
  end
```

(a) Before datatype lifting

```
datatype tree =
    Leaf of int
  | Node of tree * tree
exception Fail of int

val _ =
  let

  in
    ...
  end
```

(b) After datatype lifting

Figure 3.6: The TIR before and after datatype lifting. Exception declarations are also lifted.

```
exception Exception
fun f x =
  (raise Exception handle _ => 1) +
  (raise Exception handle _ => 2)
```

Figure 3.7: A problematic case when `handle` blocks clear the operand stack.

Consider the expression in Figure 3.7. Because the JVM is designed for an imperative language, try-catch blocks are not expressions. When an exception is thrown, the work stack is cleared. To see the importance, consider an exception thrown by + in Figure 3.8. Both operands (1 and 2) would be on the stack, but clearly any code in the catch block will not use these. Therefore, the work stack should be cleared. However, with code like in Figure 3.7, clearing the work stack when entering a catch means operand 1 is not available when we attempt to execute the addition.

This pass transforms code like that in Figure 3.7 into code like that in Figure 3.9, which solves the problem.

3.1.5 Lambda Lifting

Before lambda lifting takes place, the structure of TIR is similar to AST. The JVM only supports functions declared at the top level (MLC's representation is discussed in Section 3.2). There are several ways to remove nested functions from programs. Lambda lifting is a natural choice here: MLC deals with the issue of closure creation gracefully using currying.

A natural approach to lambda lifting is faced with some challenges surrounding the issue of closure creation. For example, a closure variable may not be in scope when a function is applied. To resolve this, my lambda lifter finds all

```java
int j;
try {
  j = 1 + 2;
} catch (Exception e) {
  ...
}
```

Figure 3.8: A `try-catch` block in Java.

```
exception Exception

fun f x =
  (let
    fun #a _ =
      raise Exception handle _ => 1
   in
     #a ()
   end) +
   (let
     fun #b _ =
       raise Exception handle _ => 2
   in
     #b ()
   end)
```

Figure 3.9: Example code after `handle` outlining. Note the introduction of `let` statements. This leaves the difficulty of creating closures to the lambda lifter.

immediate uses of a function and replaces them with partial applications of all closure variables. Figure 3.10 shows an example. Further difficulties arise in mutually recursive functions as the closures are dependent on each other. Figure 3.11 shows an example. An iterative approach is adopted to compute the minimal closures.

In addition to lifting nested function declarations, my lambda lifter lifts anonymous functions into top level functions. After this pass, all non-register declarations (declarations that cannot be put in a local JVM variable) exist at the top level.

3.1.6 Let Elimination

Bytecode methods specify stack space requirements at the function entry point. This includes work space and the number of local variables. Let elimination enables a calculation of the number of local variables.

The let elimination pass also performs pattern match elimination (described

```
val x =
  let
    val y = 1
    fun f z = y + z
  in
    f
  end

val _ = x 3
```

(a) Before lambda lifting.

```
fun #f y z = y + z
val x =
  let
    val y = 1
  in
    #f y
  end

val _ = x 3
```

(b) After lambda lifting.

Figure 3.10: An example of MLC's lambda lifter.

```
let
  val y = 1
  val z = 1
  fun f x =
    x + y + (g x)
  and g x =
    x + z + (f x)
in
  f
end
```

(a) Before lambda lifting.

```
fun f# (y, z) x =
  x + y + (g# (y, z) x)
and g# (y, z) x =
  x + z + (f# (y, z) x)

let
  val y = 1
  val z = 1
in
  f# (y, z)
end
```

(b) After lambda lifting.

Figure 3.11: An example of MLC's lambda lifter. Both y and z must be included in each call.

```
fun loop a = loop a
fun f _ =
  if false then
    let
      val x = loop 1
    in
      x
    end
  else
    let
      val z = 1
    in
      z
    end
```

(a) Before let elimination

```
fun loop a = loop a
fun f _ =
  funlet
    val x
    val z
  in
    if false then
      assign x to (loop 1);
      x
    else
      assign z to 1;
      z
  end
```

(b) After let elimination

Figure 3.12: Variables lifted out of lets. The `let`s nested within the expression are moved into a single `funlet` expression. To avoid invalid changes causing non-termination of `f`, `assign`s must be placed where variables were originally declared. The `assign` node is not SML syntax, but represents the internal expression defined in Figure 3.5.

later). This is because assigning variables in `let`s to values requires function arguments to have their values set appropriately. An assignment of:

```
fun f (x :: xs) =
  let val y = x
  in y end
```

Requires `x` is assigned before `y` is. Setting the value of `x` safely requires pattern match checking. Although approaches that do not involve merging these passes are possible, they involve leaving the tree in an invalid state between passes. This makes the ordering of passes brittle and defeats the point of separating the passes.

The let elimination pass changes function structure. All function bodies are replaced by a single `funlet` expression. This declares all variables at the beginning of the function. To avoid issues computing values that need not be computed, `assign` nodes are introduced at locations where variables were originally declared and defined. Figure 3.12 shows an example.

3.1.7 Pattern Match Elimination

Pattern match elimination is performed during let elimination. This process converts pattern matches into sequences of `if` statements.

This is implemented with backtracking. That is, if we have a function of n arguments and a pattern fails to match the k^{th} argument, the next pattern starts matching from argument 0. I have found this to be suitably efficient.

`x`: Assign `x` to the value of the parent node.

`_`: Evaluate to true.

`c`: For `c` some constant. Evaluate to true if the parent node is equal to `c`.

`[x`$_1$`, ..., x`$_n$`]`: Check that the length of the argument is n. If so, assign each element to the appropriate list element. Recursively pattern match on each list element, with list element as the parent node. Evaluate to true if all list elements match successfully.

`(x`$_1$`, ..., x`$_n$`)`: We are guaranteed by typechecking the tuple is the correct length. Assign each x_i to the i^{th} element of the tuple. Evaluate to true if the pattern match for each element evaluates to true.

`x :: xs`: Check the length of the parent node is at least 1. If so, assign `x`, `xs` the head and tail of the parent node, respectively. Then match `x` and `xs` recursively.

`Datatype(args)`: Check the datatype is the type expected using `instanceof`. If so, extract the arguments and recursively match. If there are no arguments, evaluate to true. Otherwise, evaluate to false.

Figure 3.13: Pattern match conversion algorithm. The parent node begins as the function argument. Each parent node is assigned into a variable to avoid recomputation.

Non-backtracking methods exist (e.g. Maranget [29]) and would be suitable extensions.

The algorithm I implemented is simple, but generates some tautological code. This choice was made because it is easier to reason correctness. Further, MLC has optimisation passes specially designed to remove this code bloat. The decision taken here, as elsewhere, is that lowering passes aim solely to change the representation correctly. Optimisation passes aim to make generated code faster. Keeping these aims separate is important in maintaining MLC's structure.

The algorithm by cases is shown in Figure 3.13.

3.1.8 Variable Numbering

The JVM uses naturals to identify local variables. As all local variables are declared into a single top-level `let` expression (Section 3.1.6), this pass numbers all variables 0 to n.

3.2 Representing TIR with Bytecode

In this section, rather than focusing on the details of bytecode, I present implementation details in terms of Java where possible. JVM bytecode was

```java
public class Tuple {
  public Object[] elements;

  public Tuple(int size) {
    elements = new Object[size];
  }

  @Override
  public boolean equals(Object other) {
    ...
  }
}
```

Figure 3.14: A sketch implementation of `Tuple` class.

designed specifically for Java 1.0 so the two bear close resemblance. As a result it is often easier to see what MLC generates in terms of Java.

The JVM offers features not regularly found at assembly level. It provides local variables indexed by 16-bit integers, so true register allocation is not needed. The JVM provides classes and instructions to invoke methods statically or dynamically. It provides instructions for creating objects (the only way to allocate heap memory) and garbage collection for these objects. It can load arbitrary classes at runtime. The JVM does significant runtime checking. Variables may not be accessed before they are assigned. Function calls with incorrect argument types result in Java exceptions. These features are important in my design choices. The most important parts of my representation are described:

Tuples: Tuples are represented with the class `Tuple`, defined in the MLC libraries.

My lowering plan states that tuples should be represented as in Scala, with a fixed set of classes `Tuple2, ..., TupleN` each holding a fixed number of elements.

For flexibility, I chose to represent these as a single class. `Tuple` contains an array of tuple elements. The Java code for this is simple, and shown in Figure 3.14.

Single Curried Argument Functions: Single curried argument functions are represented by classes on the JVM. This decision was again inspired by Scala. In Scala, functions are a set of classes, `Function1, Function2, ..., FunctionN`; so, no function with more than `N` arguments can be represented.

For flexibility, MLC defines a single function class that can take arbitrarily sized tuple objects as argument. All functions extend the abstract `Function` class. This specifies an `apply` method, taking an `Object` and

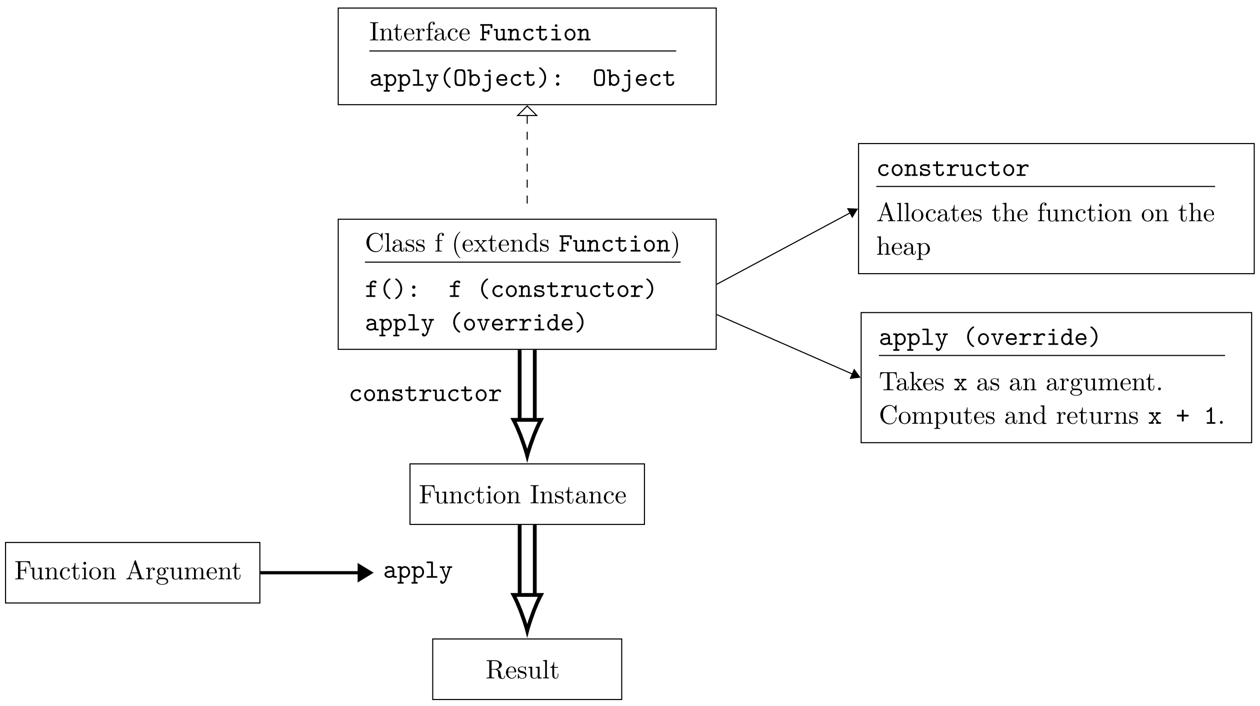


Figure 3.15: The life cycle of a single argument function `fun f x = x + 1`.

returning an `Object`. Casting is performed as required to coerce objects to the correct types.

Figure 3.15 shows how single argument functions work.

Multiple Curried Argument Functions: Multiple curried argument functions are formed with chains of single argument functions. The first curried application returns a new function. The last curried application executes the compiled function body.

A key design choice is how curried arguments are passed. One approach is, for the i^{th} curried call, to pass a reference to the parent function (the $(i-1)^{\text{th}}$ curried call). Access to the 0^{th} argument can then be achieved by traversing all the parents. However, this $O(\mathit{CurriedLength})$ access time per argument is far from optimal for arguments accessed many times.

Instead, for the i^{th} curried function call, all $i-1$ previous arguments can be passed and stored as class-local variables. The creation of a curried function closure is then $O(\mathit{CurriedLength}^2)$ but it allows $O(1)$ argument access time. This is what is implemented in MLC. This implementation better opens scope for future optimisations to include elimination of currying (instead using tuples or multiple argument functions). This is a common feature for SML compilers to have e.g. CakeML [30] and is listed as a useful pass by MLton [31]. Figure 3.16 shows how curried functions work.

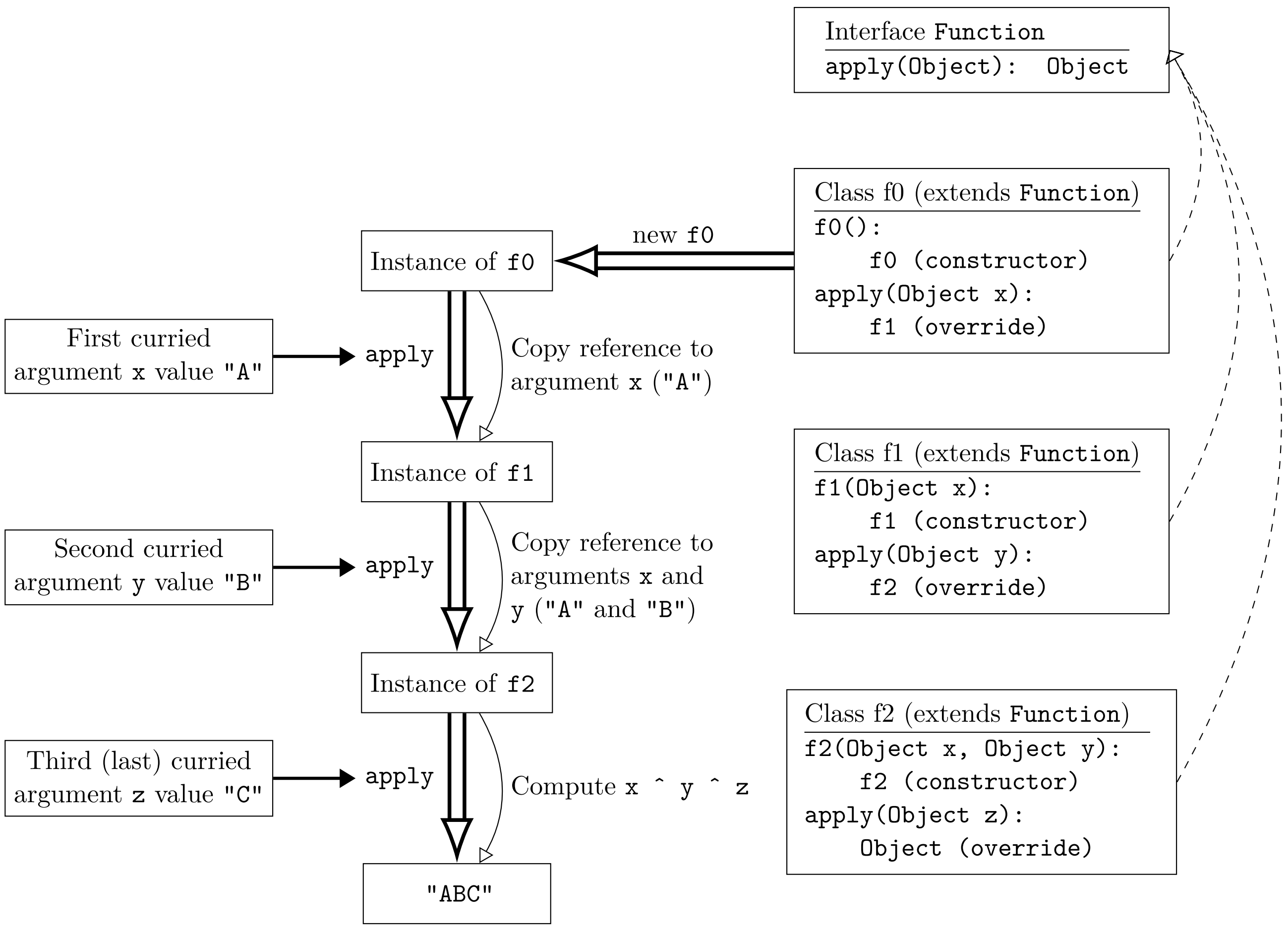


Figure 3.16: Diagram of the behaviour of an n argument curried function, `fun g x y z = x ^ y ^ z`, applied as `g "A" "B" "C"`

Expressions: Bytecode is stack oriented. Expressions are computed in the normal way for stack-oriented languages. For an n-ary operator, the code is:

```
<compute operand 0>
<compute operand 1>
...
<compute operand n>
operation
```

Entry Point: The entry point of a program is all top-level `val` declarations. These are computed within the `main` method on the JVM.

Datatypes: Datatypes are represented as Java classes. Each datatype is an instance of `Function` so implements an `apply` method. The `apply` method is used to create new datatypes, taking the arguments of the datatype and returning itself. Each datatype implements an `unapply` method, which returns the data within the type.

Pattern matching uses the `instanceof` Java primitive. This is described in more detail in Figure 3.13. Equality requires each datatype class to have a unique ID. `instanceof` may not be used like in pattern matching as it requires a statically known comparison class which is not possible in polymorphic equality functions. To check for equality, IDs are compared. If they are identical, the data in each datatype is recursively compared.

Exceptions: Exceptions are a special case of datatypes. The standard library contains an `MLCException` class that extends the `Datatype` class.

A `getThrowable()` method returns a throwable Java object. This throwable is a wrapper around the internal datatype, so the datatype can be recovered if the exception is caught.

Catch blocks catch all `MLCException` exceptions. When an exception is caught, the datatype is extracted from the throwable. Pattern matching happens on the datatype. If no matches are found, the exception is re-wrapped and re-thrown.

3.3 Optimisation Implementation Details

All optimisations are trade-offs. Any implementations must consider compile time as well generated code speed-ups. I have taken a similar approach to GCC, where optimisations must take less than $O(n^2)$ time, where n may be any programmer-visible feature [32].

3.3.1 Tail-Call Elimination Pass

My tail-call elimination (TCE) pass has two parts. First, tail recursive functions are identified. Next, tail recursion is replaced with a `while(true)` loop. Semantics for this loop are given in Figure 3.5.

Elimination consists of four steps:

```
fun f x = x

fun count n = (* A *) f ((* B *) count (n - 1))

val x = (* C *) count 10
```

Figure 3.17: Classification of call sites. (A) and (B) are labelled as hot, (C) is labelled as cold.

---

1. Function parameters are made mutable.
2. The function body is replaced with a `case` statement. Pattern matches in the function definition are recreated within the `case` statement.
3. Any return values are replaced with `break(value)`.
4. Any recursive calls are replaced with reassignments to the function parameters followed by a `continue`.

In order to convert function patterns into `case` patterns, TCE introduces more tuples. Curried function arguments are tupled for pattern matching. This introduces some overhead. Nevertheless, TCE is a worthwhile pass to increase the expressiveness of SML.

3.3.2 Function Inlining

At each function call site, the inlining pass finds the called function definition. The inlining size trade-off is estimated, and the function is inlined if profitable. Decisions are made based on the nature of the call site and the size of the function.

My approach is simple due to time constraints. Call sites are identified as *hot* or *cold*. A hot call site appears in a recursive function. All others are cold. Figure 3.17 shows an example.

Estimating the cost of an inlined function is difficult. Inlining is ultimately an optimisation that should occur early because it enables many further optimisation passes. The shape of an inlined function may change considerably before compilation is finished, because many passes have yet to run. A function call takes 4 instructions in bytecode. Experimentation suggested 28 instructions as a suitable maximum to inline. This is a conservative estimate, and a justified one since inlining can easily go wrong. My pass estimates function size by counting the number of tree nodes.[5]

This can add significant compilation time due to increased code size. Figure 4.10 shows an example. The time complexity of this pass is:

$$O\left(\text{Tree Size} + num_{\text{Call Sites}} \times \text{Curry Depth} + num^2_{\text{Functions}}\right)$$

[5]This has obvious failure cases, e.g. functions with dead code that is later eliminated.

Unfortunately, this equation has a large constant factor. With a better implementation,[6] the addition factor of $num^2_{\text{Functions}}$ could be reduced to $num_{\text{Functions}}$. Note that since an N argument curried call is applied at N call sites, this pass is in fact quadratic in Curry Depth. Overheads, the 3 tree walks required and the poor inlining decisions risk mean inlining must be requested with a command line flag.

3.3.3 Copy Propagation Pass

Elimination of pattern matching introduces many superfluous variable copies (see Figure 3.13).

This pass is flow insensitive. It works by finding variables with single assignment sites. It then only considers copying assignments. Uses of that variable may be replaced by the assignment's r-value. This pass has also been adapted for constant propagation where variables assigned to simple constants are copied into use sites.

The flow insensitivity is justified as no language features which could take advantage of flow sensitivity are supported. All input code is pure so there is at most one assignment to each variable. All compiler generated copies may be eliminated in a flow insensitive manner.

Beyond simplifying the implementation, this makes copy propagation significantly faster by avoiding constructing a CFG (control flow graph). I have previously discussed my project's aim to provide a (functional) language-independent compiler. To support this aim, the copy propagation pass is designed so use of flow-sensitive approaches should not require a total rewrite.

This pass has time complexity (per function):

$$O\left(\text{Variables} \times \text{Function Size}\right)$$

Where Variables includes variables generated by MLC. The time complexity is explained by the following algorithm:

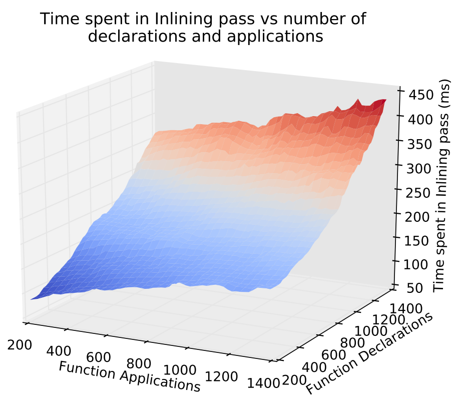
1. Iterate over each variable ($O(\text{Variables})$). For each:
    (a) Find assignment sites for a variable ($O(\text{Function Size})$).
    (b) If copy propagating, find all uses of the variable and replace them ($O(\text{Function Size})$).

Measurements show this equation to be reasonable. Experimental data are shown in Figure 3.18.

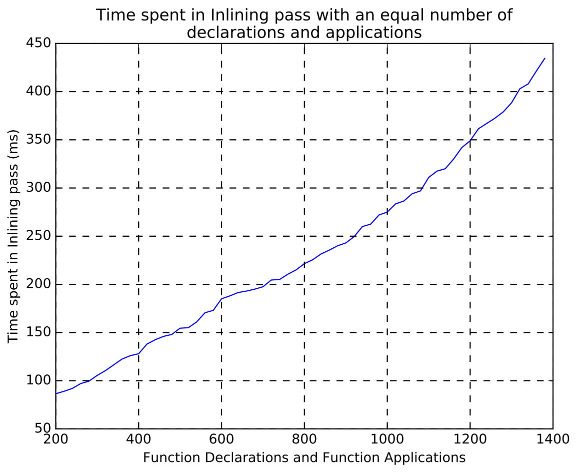
3.3.4 Simplify Passes

The simplify passes are critical for compiler performance. They enable simplifications in lowering and optimisation pass design. Some implemented simplifications are shown in Figure 3.19. The passes are designed so that adding

[6]This could be achieved using hash tables rather than linear searches to find function definitions.



(a) Time spent in the inlining pass vs the number of declarations and applications.



(b) This shows a cut through Figure 3.18a along the line where Function Applications = Function Declarations.

Figure 3.18: Execution times plotted against the number of functions and applications. Due to time constraints, a single data point has been taken at each mesh location. Values have been smoothed with an 8x8 median filter.

```
(* Before simplication.  *)
case x of
      n => n + 1

(* After simplication.  *)
x + 1
```

(a) Simplifying a case statement.

```
(* Before simplification.  *)
if true then
  x
else
  y

(* After simplication.  *)
x
```

(b) Simplifying an `if` statement.

Figure 3.19: Examples of the simplify pass.

```
val a = 1
val b = 2
val c = 2 * a
val d = print(Int.toString(a + b))
```

Figure 3.20: The stores to `a` and `b` are not safe to remove. The store to `c` is safe to delete. Removing the write into `d` is safe but removing the computation is not.

---

additional simplifications is easy. The defining feature of an optimisation in a simplify pass is contextual independence. Any structure that can be reduced without program-wide context fits in the simplify infrastructure.

The simplify passes walk the tree and pattern match on elements. Worst case runtime is nearly linear:

$$O\left(\text{Tree Nodes} \times \text{Pattern Size}\right)$$

Where we rarely go beyond $O\left(\text{Tree Nodes}\right)$ as matching patterns completely is rare. Overhead is small because simplify requires a single tree walk and no additional data structures.

3.3.5 Dead Store Elimination Pass

Dead store elimination removes unused computations and stores to dead variables. Dead variables will be written to before they are next read, or will never be read from.

The pass uses the standard algorithm described in the optimising compilers lecture course [33]. A CFG is built, and liveness information is determined. Dead stores are not computed if their computation is safe to delete. If their computation is not safe to delete, the value is computed and thrown away. Figure 3.20 shows an example.

Obviously, this paradigm does not occur in pure functional languages without reassignments. The need for this pass is justified by JVM runtime checks. The JVM requires stores to all variables before they are read. This is asserted on all statically visible code paths. Elimination of pattern matching introduces tautological expressions where dynamically unavailable but statically-visible paths do not assign before use. To avoid JVM verification errors, lowering from TIR to ByteR requires defensively inserting stores of `null` to local variables upon function entry. Most of these stores can be safely eliminated. Dead store elimination exists in the ByteR optimiser for exactly this. This pass is named `dse` in the compiler due to a misunderstanding in the proposal.

3.3.6 Peephole Pass

The peephole pass implements a linear walk of instructions. A handler class requests all sequential sequences of instructions with length n, for every required

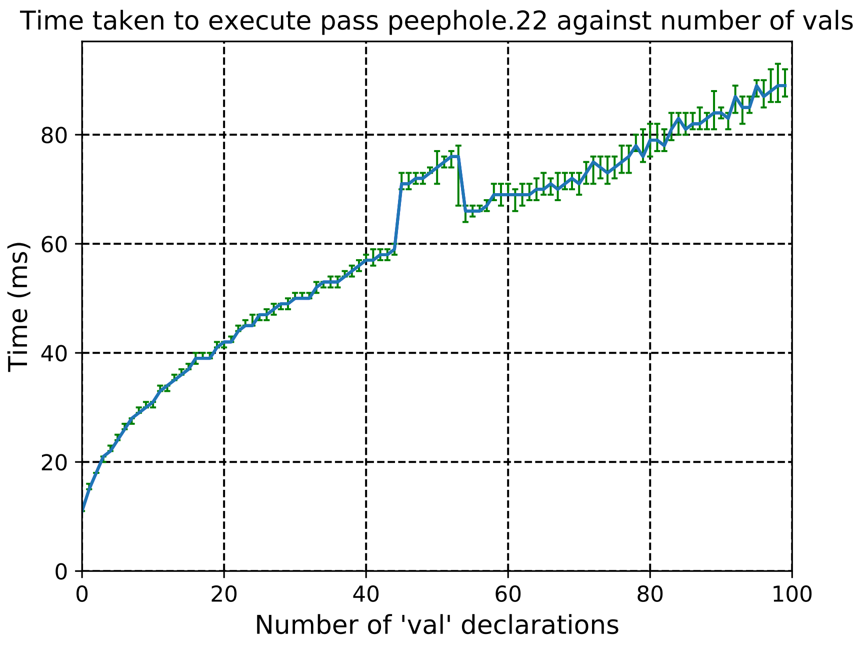


Figure 3.21: Time taken to execute the peephole pass plotted against the number of `val` declarations. The spike between 45 and 53 comes from the garbage collector. Enabling verbose printing on the garbage collector results in "out of memory" messages during peephole for between 45 and 53 `val` declarations. More `val` declarations and faults occur before. Fewer `val` declarations and faults occur after. All `val` declarations exist in a single bytecode method. The growth appears linear after initial faster growth.

---

n. Each sequence is matched against defined peepholes, each of which extend a common `Peephole` interface. On a match, the peephole returns new instructions to replace the old ones.

Currently, there are relatively few peepholes implemented. These focus on removing common `push`-`pop` sequences and variable boxing followed by variable unboxing.

The handler is designed to manage a large number of peepholes using a binary search of peepholes on each sequence. A linear search is currently used because the number of peepholes is small enough that the extra overhead of binary search does not justify the reduced time complexity. Peepholes provide an easy way of making simple changes. Additional peepholes are best identified by inspecting generated code for common sequences of instructions that should be removed or replaced.

With this extension, the time complexity of the peephole pass becomes:

$$O\left(\sum_{\text{Functions}} \log\left(\text{Num}_{\text{Peepholes}}\right) \times \text{Instrs}_{\text{in function}} \times \text{Biggest Peephole}\right) \quad (3.1)$$

Since the set of peepholes is fixed on each compilation, this is a linear optimisation pass. The measured time of this pass is shown in Figure 3.21.

The peephole pass is run by default. This is for two reasons: first, the time complexity in Equation 3.1 is linear in program size. This is not an asymptotic increase in compile time compared to lowering passes that must run over all instructions. Second, the TIR-to-ByteR lowering pass produces many detrimental compilation artefacts. This pass removes those. I have concluded that while this is strictly an optimisation, the performance benefit for the low time complexity means it should not require an optimisation flag to enable.

3.3.7 Phase Ordering Problem

Ordering of optimisation passes has drastic effects on generated code quality. Where passes act on the same parts of a program, well-reasoned decisions about their order are important. Intuition can be gathered both from MLC's internal structure and from existing compilers. I will discuss pairwise orderings of optimisation passes that interact.

Tail Call Elimination and Inlining: Experiments with LLVM[7] and inspecting the MLton source code [34] show they both do tail-call elimination before inlining.

I believe tail-call elimination would be hampered if inlining ran first. To see this, consider a small function that can be inlined, but is marked for tail recursion. The tail recursion will be removed, but the old version may have been inlined elsewhere. This will leave one tail call that could have been eliminated. An example of this redundant function call is shown in Figure 3.22.

Simplification and Inlining: The best inlining decisions introduce many simplifiable structures. A following pass simplifying these is extremely profitable.

Figure 3.23 shows an example.

Copy Propagation and Simplification: Copy propagation detects and removes copies generated by lower program.

However, copy propagation fills the tree with redundant structures. To maintain tree correctness, eliminated copies are replaced with `unit` (they are assignments to mutable variables). By deferring simplification until after copy propagation, MLC can remove these long before the peephole pass runs. This is beneficial for compile time and helps following passes understand the tree. An example is shown in Figure 3.24.

[7]This was done with the `-print-after-all` flag.

```
(* count before TCE.  *)
fun count 0 = 0
  | count n = count (n - 1)

(* count after TCE.  *)
fun count n =
  while true (
    case n of
         0 => Break(0)
       | x => assign n to x - 1; Continue
  )

(* f before inlining.  *)
fun f x = count x

(* f after inlining the TCE version of count.  *)
fun f x =
  case x of
       n' =>
        while true (
          case n' of
               0 => Break(0)
             | x => assign n' to x - 1; Continue
        )

(* f after inlining the original version of count.  *)
fun f x =
  case x of
       0 => 0
     | n' => count (n' - 1)
```

Figure 3.22: An example for why tail-call elimination should run before function inlining.

```
fun add (x, y) =
  x + y

fun f(x, y) =
  case (x, y) of
       (x', y') =>
          x' + y'
```

(a) After inlining of `add` into `f`.

```
fun add (x, y) =
  x + y

fun f(x, y) =
  x + y
```

(b) After Simplification.

Figure 3.23: Code before and after `case`-elimination from the first Simplify pass. Here, the function `add` has been inlined, resulting in a redundant case statement.

```
val _ =
  assign x to 1;
  assign y to x;
  assign a to 2;
  assign z to x;
  y + z
```

(a) Before copy propagation.

```
val _ =
  assign x to 1;
  ();
  assign a to 2;
  ();
  x + a
```

(b) After copy propagation.

Figure 3.24: An example of copy propagation leaving defects (empty tuples) that can be cleaned up.

```
iconst_1
invokestatic Method java/lang/Integer valueOf
    (I)Ljava/lang/Integer;
astore_1
```

(a) Before DSE.

```
iconst_1
invokestatic Method java/lang/Integer valueOf
    (I)Ljava/lang/Integer;
pop
```

(b) After DSE concludes the store is dead.

Figure 3.25: The DSE pass can detect that the store to local 1 (`astore_1`) is dead. However, it cannot detect that the function call is safe to remove. It replaces `astore` with `pop` (in red). The peephole pass knows the function call is side effectless and can be deleted.

Dead Store Elimination and Peephole: Dead store elimination (DSE) runs before peephole. This is because DSE creates opportunities that peephole can act on. This is rarely true the other way around.

For example, stores the DSE pass can detect as dead but cannot eliminate (e.g., because the value is generated by a function call DSE cannot prove is side effectless) are replaced with `pop` instructions. An example is shown in Figure 3.25. The peephole pass is better at detecting these cases as it has a list of functions known to be side effectless.

This is not because it is impossible to analyse side effects within the DSE pass. However, leaving this task to the peephole pass removes duplicated code and increases maintainability.

3.4 Testsuite Implementation Details

The core compiler alone is over 11,000 lines of Scala. Any sustained progress requires that regressions are tracked as they are introduced. MLC's testsuite aims to solve this problem. The testsuite is implemented in two parts, a Python script to run tests and a set of SML files to build. The script provides options for commonly required actions during development. When run, it searches test folders for SML files and compiles them. The output is a list of all tests run, with failures marked.

Each SML file contains several directives specifying how the test is executed and what result is expected. There are four directives:

`t-compile:` Compile this file. Compilation flags may be specified.

`t-fail:` Expect compilation to fail (e.g. due to a type error).

`t-run:` After compilation, run the executable. Accept a parameter to verify expected output.

`t-scan:` Match a specified regular expression against a specified dump file and report the result.

Dump files are optionally produced by MLC during each compilation pass. They show internal state after the pass runs, and useful debugging information (e.g., the dead store elimination pass shows liveness information). The compiler produces a dump file when a flag is specified. Variants of the `t-scan` directive can require expressions to appear no, one or more times, or N times for any particular N.

This approach has worked extremely well. There are no mutable-state issues since a difference instance of MLC runs each test. Regression tests are easily added by putting the SML file that failed into the testsuite.[8] The whole testsuite is wrapped in a shell script. This builds the compiler, the standard libraries and runs the testsuite. TravisCI runs this on every commit [15]. This approach makes it possible to use entire programs for thorough testing. The testsuite currently contains 212 SML files.

[8]This is the case provided there are no licence issues.

3.5 Benchmarking Implementation Details

The benchmarking is similarly composed of two parts: the Python run script and a set of benchmarks. The run script runs the benchmarks with appropriate flags, collects performance information and verifies output is correct. Performance information includes produced executable size and execution time.

In addition, the Linux tool `perf` [11] is used to gather detailed statistics for each benchmark. The options for `perf` are machine dependent, but on my benchmarking machine I collect:

- Instruction count.
- Branch predictor hits and misses.
- Data cache hits and misses.
- Instruction cache hits and misses.
- Clock cycles spent in the CPU, on the bus and in the memory system.

These data are useful in pinpointing how — and whether — optimisations help. The Linux tool `taskset` [35] is used to avoid noise problems introduced by migration between cores.

Recorded data are stored in a JSON format compatible with the LLVM Nightly Testing [20] (LNT) framework. LNT is used to track executable size and speed through time. Continuous visualisation of performance data is useful in making development decisions, and in providing a basis for the evaluation chapter. Only benchmarks written in the first phase of this project were tracked with LNT.

The benchmarks are:

alignment: Computes a sequence alignment between genomes with the Smith-Waterman algorithm [36]. Tracked with LNT.

dft: Computes a discrete Fourier transform. Tracked with LNT.

mandelbrot: Computes the Mandelbrot set. Slightly adapted for my project from Cumming [19]. Tracked with LNT.

matrix_multiplication: Multiplies matrices. Tracked with LNT.

inlining: Runs many simple function applications designed to showcase inlining. Tracked with LNT.

utils: Computes several small functions from ML For the Working Programmer [37]. Tracked with LNT.

knuth_bendix: Implements the Knuth-Bendix algorithm [38]. Taken from MLton's benchmark suite.

life: Simulates Conway's game of life as presented by Gardner [39]. Taken from MLton's benchmark suite.

peek: Stress tests lists and datatypes. Taken from MLton's benchmark suite.

tak: Computes Takeuchi's `tak` function, designed to compare LISP systems as presented by McCarthy [40]. Taken from MLton's benchmark suite.

3.6 Standard Library Implementation Details

A complete standard library is not the aim of this project. One of the most important extensions I propose is enabling access to the Java standard libraries.

Nevertheless, some libraries are required to represent tuples, higher order functions and immutable lists. These are implemented in Java. They are compiled during installation and are statically linked into produced `jar` files.

The libraries provide maths functions required to run my benchmarks. The libraries have been implemented in a manner compatible (with the same observable behaviour when running my benchmarks) with the MosML, MLton, PolyML and SML/NJ libraries so benchmarking results may be compared.

3.7 Compilation Scripts Implementation Details

As discussed in the introduction to this chapter, MLC is not straightforward to use without a wrapper.

I have provided an installation script to deal with the complexities of installing the standard libraries. This installation script also generates a shell script to hide auxiliary and mechanical compilation steps (such as running the assembler and linking the `.jar` file) from the user.

The installation script builds MLC and the standard libraries. It then downloads the assembler. Finally, it creates links to the assembler and standard libraries in the execution script. The flow for the execution script is:

1. Create a temp directory for generated class files and the bytecode assembly file.
2. Execute MLC with command line arguments forwarded.
3. Execute Krakatau on the assembly file. This produces a number of `.class` files.
4. Create a `jar` file from the `.class` files and libraries.

3.8 Summary

There are three major components of this project: the compiler, the optimiser and the infrastructure. All three integrate cleanly with each other despite cooperation requirements.

I have been successful in modularising MLC at several scales. The testsuite and benchmarking suite are compatible with other compilers. This has been demonstrated in practice by running my testsuite with MosML, and running the benchmarking suite with MosML, MLton, PolyML and SML/NJ. Internally, decoupling of components has been mostly successful. The TIR representation is as generic as planned. However, in retrospect, the ByteR representation is not as flexible as intended. Ideally, optimisation passes written for ByteR would be flexible enough to adapt to any target. This has not turned out to be the case. I believe the current bytecode optimisations are only easily extendable to other stack-based targets with similar restrictions on jumps as bytecode. To add another backend, I would first focus on a single generic representation decoupled from the output language and only converted to target instructions on output.

Chapter 4

Evaluation

4.1 Methodology

The methodologies behind each analysis share much common ground. Multiple data points have been collected when possible (where not this is made clear). The upper and lower approximately[1] 15th percentiles have been used for error bars.

I have used a median-based approach. Minimum-based approaches better model best-case execution time, as the lowest runtime corresponds to the one with least interference. However, a central measure better reflects real-world use cases. This approach is taken by SPEC [12]. Using a median-based approach works under the assumption that distributions are unimodal. To verify this, all plotted data points have been manually observed to adhere to a unimodal distribution by observing cumulative frequency graphs. Data that showed bi-modality were recollected.

Benchmarking data have been collected on a machine provided by the Computer Labs. The machine runs Ubuntu 14.04 and has an Intel Core i5-5200U CPU running at 2.20 GHz. The JVM is Oracle's Java SE JVM. Benchmarking has been performed with a compiler built from commit `a14879b`. This evaluation focuses on a fixed version of MLC, so data collected for LNT have not been used.

4.2 Execution Time

Execution time is a central goal of my project. Beyond this, I will explore how individual passes contributed to speed-up and how they interact. MLC will also be compared to other ML compilers.

Figures 4.1 and 4.2 show the effect of individual passes on execution time, with and without the JIT enabled respectively. The benchmarks are discussed in Section 3.5.

[1]Some analyses take too long to gather this much data. In these cases, error bars are the first datum within the 15th percentile.

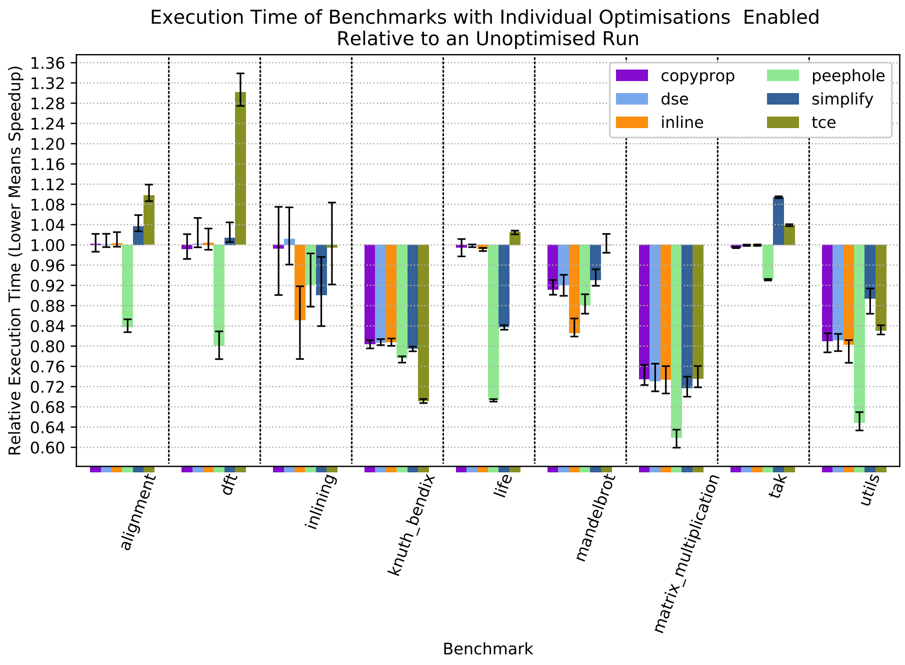


Figure 4.1: Execution time of benchmarks with individual optimisations enabled relative an unoptimised run. The further down a bar is, the more speed-up it provided. This figure shows the contribution of optimisation passes with the JIT enabled. The JVM masks several optimisations (performing them easily) such as copy propagation and dead store elimination. Their full effects are not seen here.

In Figure 4.1, the increase in execution time with tail-call elimination enabled for some benchmarks is particularly interesting. The JVM used on my benchmarking machine is the Oracle Java SE JVM. In Figure 4.1, performance decreases by approximately 10% and 30% for `alignment` and `dft` respectively. Using a separate machine with the OpenJDK JVM, tail-call elimination increases runtime performance of `alignment` by approximately 40%, and of `dft` by approximately 15%. As a result, I have concluded this result is JVM specific. This is discussed more with reference to Figure 4.2 below. Here it is safe to conclude that the Java SE JVM missed an optimisation.

The other feature that stands out from Figure 4.1 is the noise in the `inlining` benchmark. The benchmark appears to exhibit intrinsically noisy behaviour, possibly due to JVM optimisation decisions. However, there are cases where noise is low enough to draw conclusions.

Several optimisations appear to have small effects in 4.1, such as copy propagation in many benchmarks. This is because the JVM is able to perform these optimisations easily in most cases. Further, the eliminated instructions in these cases cost little per instruction on the JVM. These optimisations do have noticeable positive effects. As can be seen in Figure 4.16, there are a number

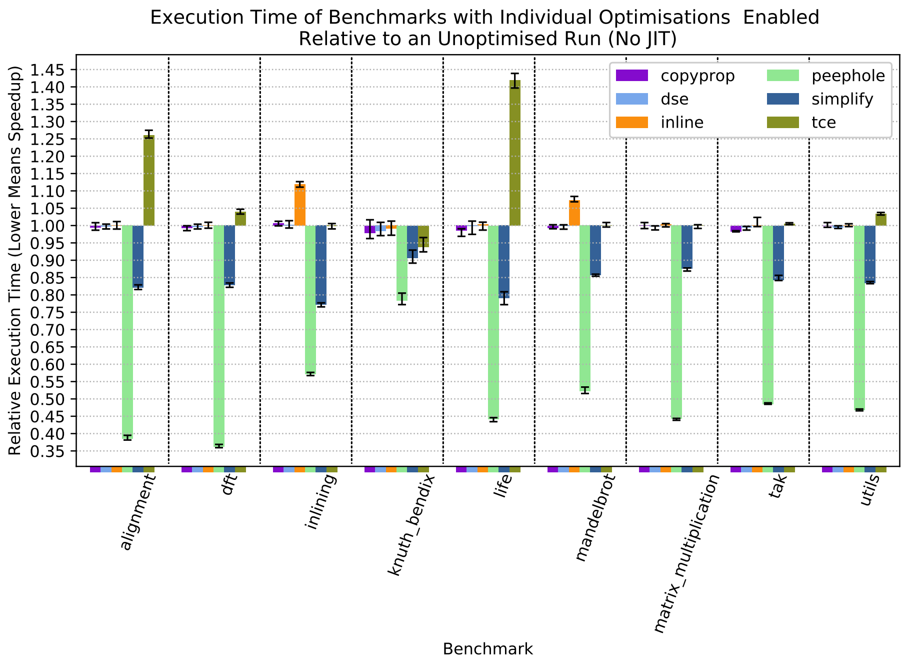


Figure 4.2: Execution time of benchmarks with individual optimisations enabled relative to an unoptimised run without the JIT. The JIT has been disabled here. This provides context of how much each optimisation pass helps individually (instead of with JVM optimisation passes).

---

of copies being removed.

Figure 4.2 shows the direct effect of passes (without JVM optimisations). Notice `inlining` and `tce` have negative effects. This results from the trade-off made between curried function applications (removed) and pattern matching on tuples (inserted). This trade-off is shown in Figure 4.3. This performance hit does not make `tce` or `inlining` bad optimisations. `tce` provides expressivity. We will see below `inlining` enables many further optimisations so inspecting individual performance is not representative.

Another feature of note is how much `peephole` helps. The `peephole` removes variable boxing and unboxing. This helps beyond removing slow code manipulating the heap. It also vastly reduces the amount of garbage to collect.

In several cases, passes were written to complement each other. This was briefly discussed with the phase-order problem in Section 3.3.7. Figure 4.4[2] shows optimisation passes interacting pairwise with each other. Notice the overlap of `simplify` and `peephole` in terms of what they optimise. This is because

[2]Some PDF viewers, such as Preview on Mac, fail to display this figure correctly. Adobe Acrobat displays this figure correctly.

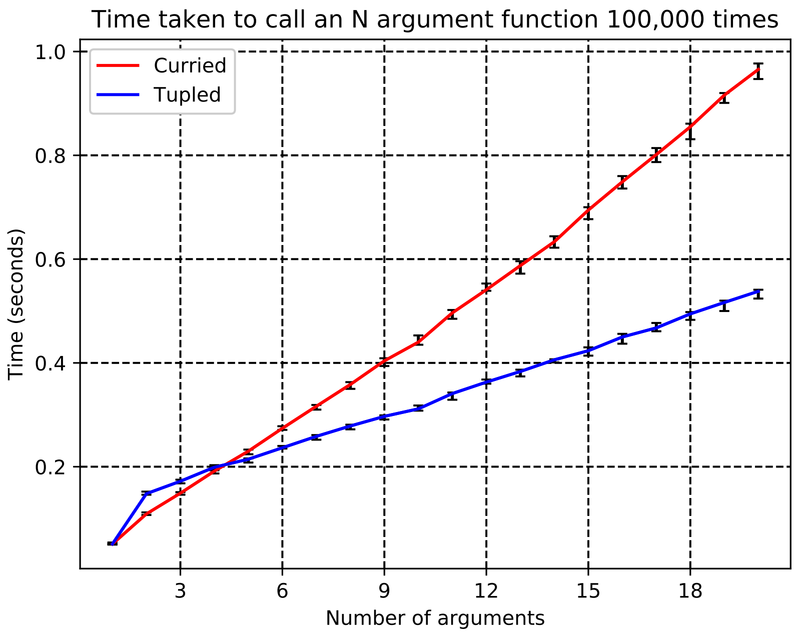


Figure 4.3: Time taken to call an N argument function $100,000$ times.

`simplify` removes many structures that `peephole` can greatly simplify.

It is clear how much inlining benefits from the presence of other optimisations. This is best understood by considering that inlining makes the code messier. Unlike a human inlining a function, `inline` leaves redundant case statements, introduces variable copies and so on. This behaviour creates opportunities for other optimisation passes.

The interaction of `copyprop` and `dse` is explained as `dse` removes initialising stores to variables. When `copyprop` runs, there are fewer variables to initialise so `dse` can do less.

`simplify` and `copyprop` also have some overlap. This is because `simplify` removes some structures that generate variable copies `copyprop` could eliminate.

Finally, it is important to see how MLC compares to existing ML compilers. This is shown in Figure 4.5. For the most part, relative performance is clear. MLC outperforms MosML, but is outperformed by PolyML and SML/NJ. MLton outperforms all by far.

This is due to more extensive optimisations and the performance benefits of machine code over Java bytecode. I did not have time to implement many optimisations (such as continuation-passing style) that I suspect would enhance performance. Further, my tuple representation described in Section 3.2 trades performance for flexibility. A tuple with a single initialising function call would be more efficient. This was the right decision at the time due to rapid project evolution. I suspect that some small tweaks could significantly increase perfor-

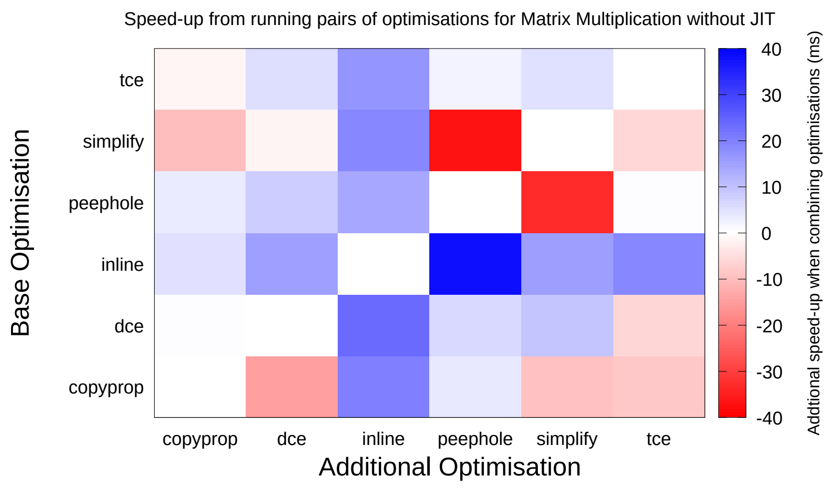


Figure 4.4: How passes interact pairwise for the `matrix_multiplication` benchmark. Speed-up is measured relative to the sum of speed-up from both passes individually. White indicates the passes are orthogonal. Blue indicates one pass complements the other. Red indicates the passes overlap in their optimisations. `matrix_multiplication` was chosen to showcase this as all optimisations have similar effect on execution time (see Figure 4.1). Asymmetry is due to noise.

---

mance.

`perf` has been used to gather hardware performance counters from benchmarks. This allows detailed investigation of how passes interact with the underlying microarchitecture. This is particularly important when the JVM's interpretation of instructions can be opaque. The goal of this discussion is to explore the effect optimisations have on generated code rather than on execution time. Further, as discussed above, the JIT interacts with programs in difficult-to-analyse ways. For these reasons, this section focuses on data collected without the JIT enabled.

Looking at Figure 4.6, we can see most optimisations reduce the number of loads. This is unsurprising. As discussed previously, `tce` and `inlining` results are explained by extra tuple pattern matches. The decrease in loads shown by other optimisations is intuitive: each reduces the work to do by removing code containing loads.

Optimisations that reduce the miss rate do so by removing heap references, where loads are likely to miss. `peephole` reduces the number of boxed types allocated on the heap. `peephole` has the largest effect due to sheer number of boxes removed.

Arguably, the more interesting effects here are increased cache miss rates from `dse`, `copyprop` and `simplify`. These increase the cache miss rate because many of the removed loads are “easy” (they always hit). For example, as discussed in Section 3.3.5, `dse` removes many unneeded variable initialisations. All have the same constant stored into them. Loading this constant many times results in cache hits from temporal locality. Likewise for `simplify`, many eliminated regions of code have high temporal locality (e.g. repeated loads of the constant 1). Likewise, `copyprop` removes many references to local variables on the stack that have spatial locality. Eliminating these results increased cache miss rates. `tce` is a special case. As discussed previously, it increases heap use, so loads and miss rate increase for the same reason that `peephole` decreases the number of loads and the miss rate.

There are similar results in Figure 4.7. `peephole` removes heap-manipulating code, which is evidently hard to issue out of order. `tce` has the inverse effect for this reason. `simplify` removes tautological code, which is by nature extremely easy to issue out of order. The `copyprop` and `dse` passes tend to remove parallel code (e.g. variable initialisations and unchained copies) although `copyprop` sometimes removes sequential copies.

The effect of inlining here is somewhat surprising, but is explained by the lack of function calls. Although allocating tuples requires more stress on the memory system, it is likely that fewer dependencies exist between instructions without function calls.

There are some conclusions to be drawn here. This analysis helps classify implemented optimisations into categories based on how they behave. These graphs clarify `simplify` acts as expected, removing significant tautological code, and `dse` removes the stores it is expected to eliminate. They allow for helpful optimisation groupings. They allow for better understanding of what hardware is most exercised under each transformation. Particularly with no clear one-to-one mapping from bytecode instructions to machine code instructions, this analysis helps us to understand what is and what is not useful.

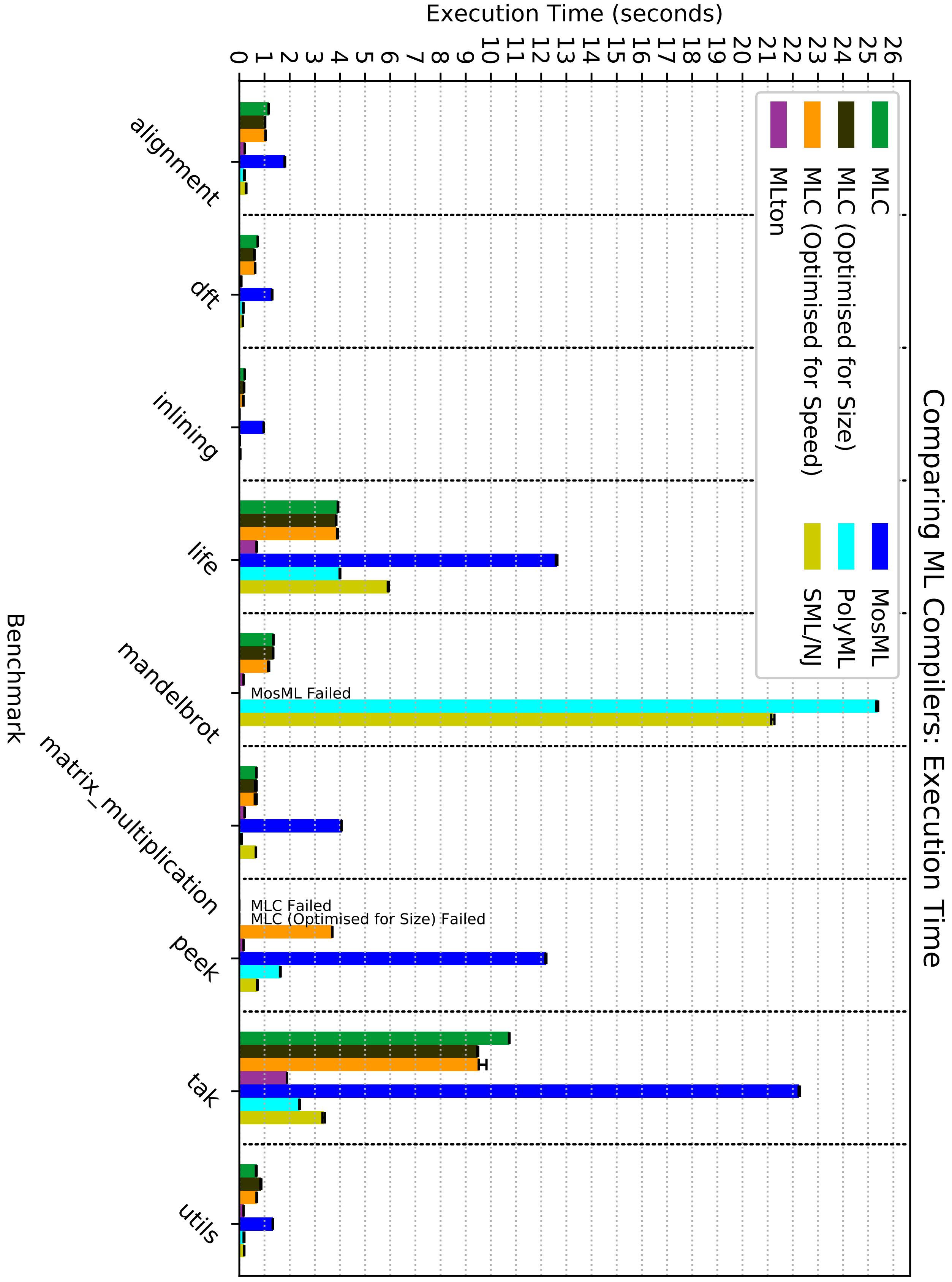
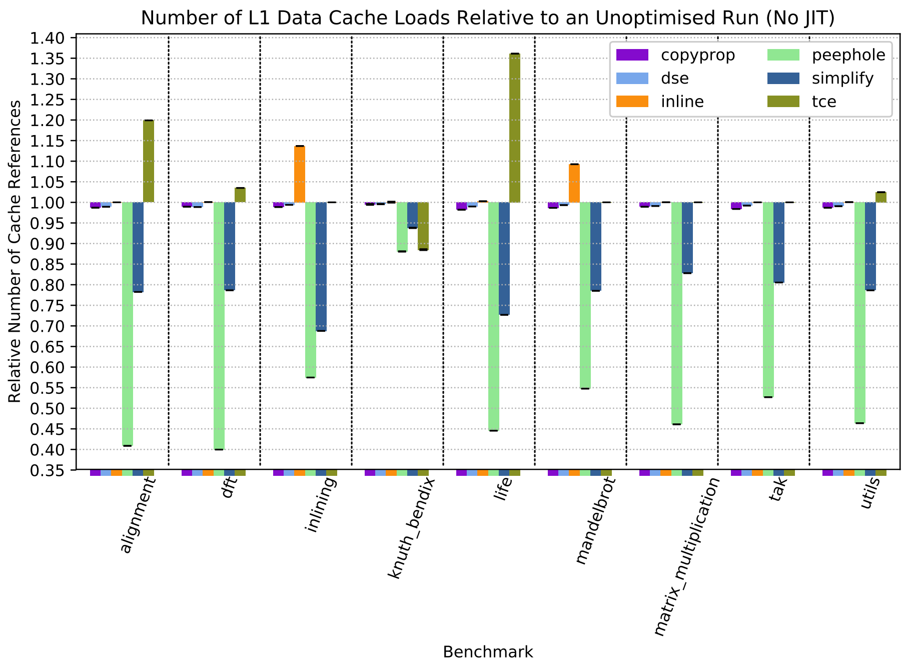
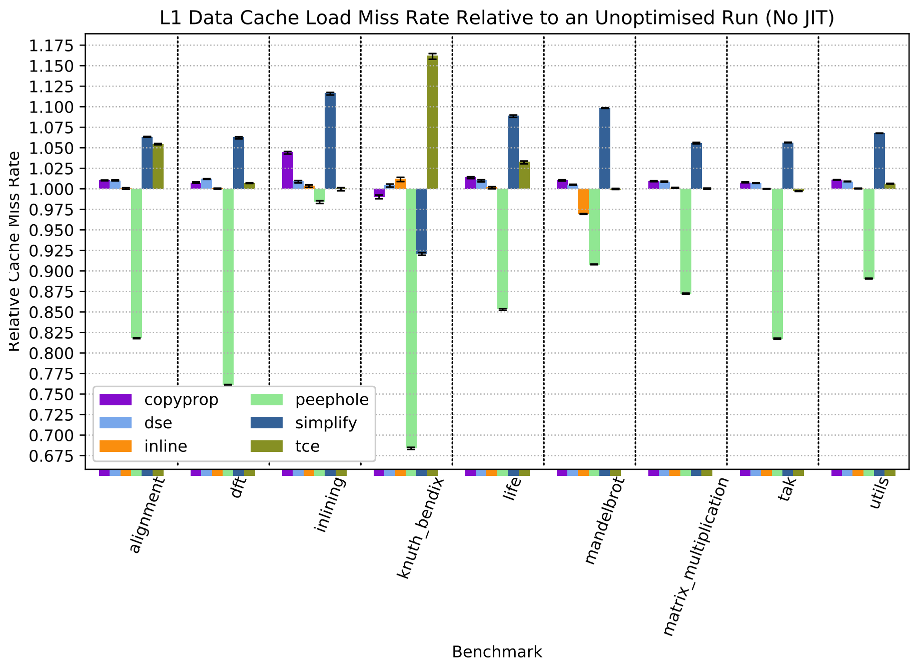


Figure 4.5: Comparing ML compilers in execution time. MosML fails `mandelbrot` due to floating point overflow, which is treated as an exception. MLC fails `peek` without tail-recursion elimination due to stack overflow. The unoptimised run of MLC includes the `peephole` pass as discussed in the implementation.



(a) Number of L1 data cache loads relative to an unoptimised run without the JIT.



(b) Data cache load miss rate relative to an unoptimised run without the JIT.

Figure 4.6: Relative L1 data cache misses under each optimisation. The load miss rates should be interpreted with reference to the number of loads.

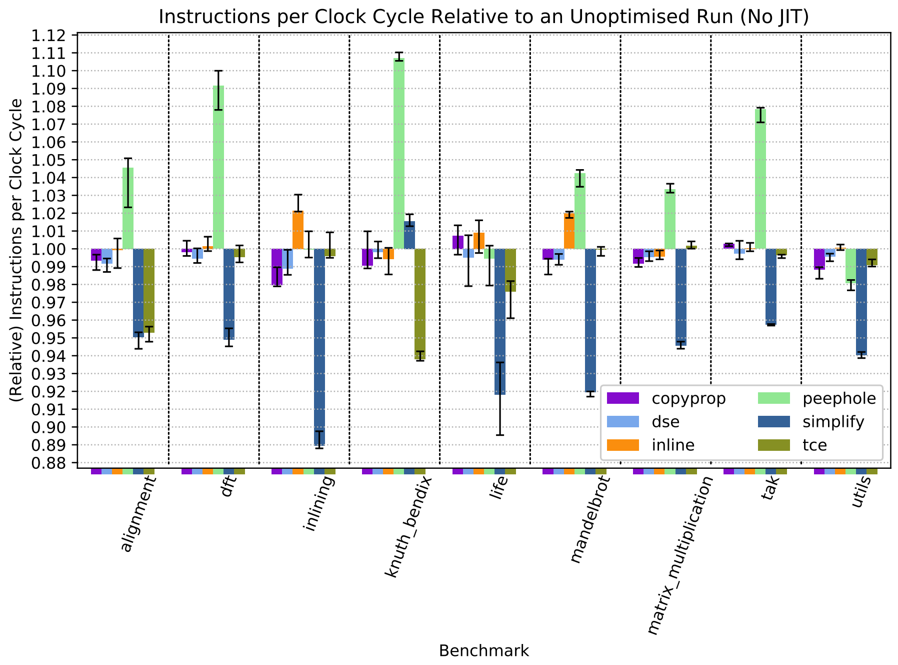


Figure 4.7: Instructions per clock cycle relative to an unoptimised run for individual optimisations. Benchmarks were run on a 4-way OoO processor. More issued instructions means more ILP. Inlining is too noisy for meaningful ILP measurements.

4.3 Compile Time

My project aims to provide a compiler fast enough to be usable. Figure 4.8 shows MLC achieves this. Compile times are large but not unheard of for ML compilers (cf. MLton).

Compilers scale well across large projects where they can run on multiple files individually and link results together. The Linux `make` [41] tool, for example, takes advantage of this. Running multiple instances is a suitable way to increase MLC's throughput.

Beyond assessing benchmark compile times, I have attempted to ensure MLC scales in different program features. Figures 4.9, 4.10, 4.11, 4.12, and 4.13 show this has been achieved (note these do not include linker and assembler times).

In all cases, the majority of time is spent in the AST phase. Within this, typechecking dominates. In large part this is due to late bug fixes. I discovered unifiers must be applied at unanticipated places within a program. More importantly, some unifiers are applied many times.

Figure 4.14 shows how time is spent compiling benchmarks. This shows that real code compiles in reasonable time, and that the assembler and linker do not excessively add compilation time.

4.3.1 Typechecker

To simplify my approach (described in Section 3.1.1), I implemented a single directional map. This means unifications walk the entire type environment. A fix I anticipate would vastly reduce compile time is using another hash map from type variables to the set of variables that have types containing this particular type variable.

The type checker behaves asymptotically as expected (although it reaches asymptotic limits faster than other compilers). The compilation time of a program with an exponential number of type variables is shown in Figure 4.15.

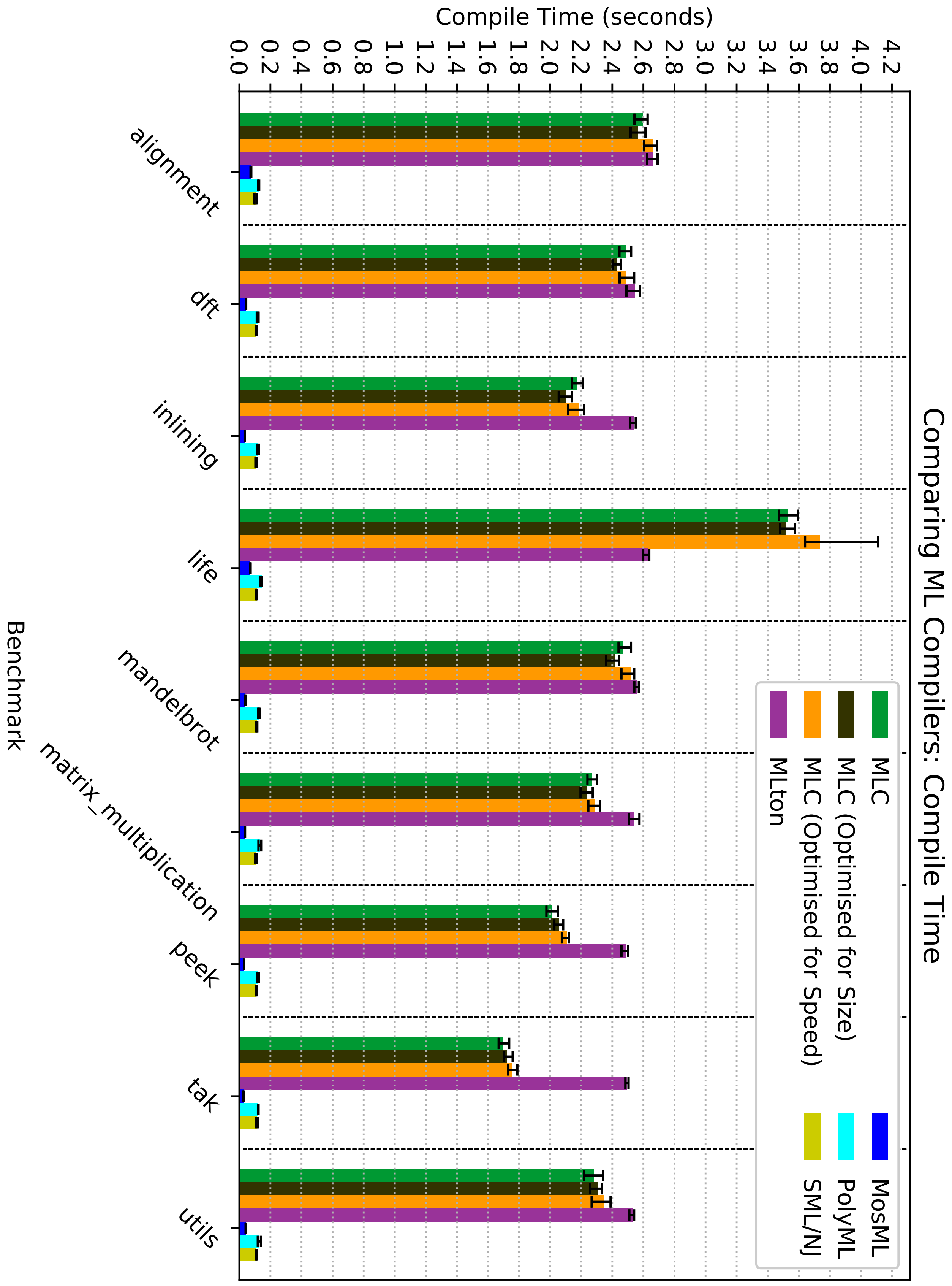
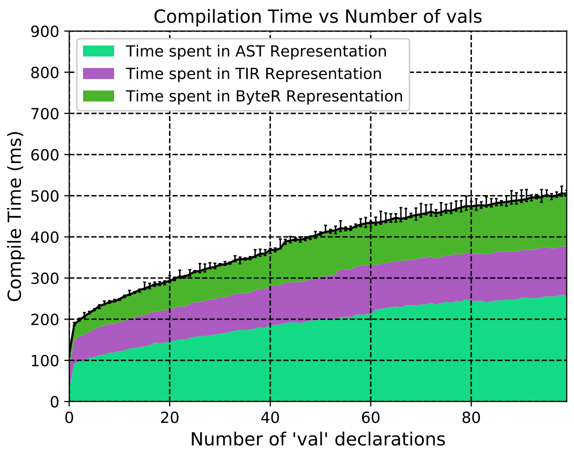
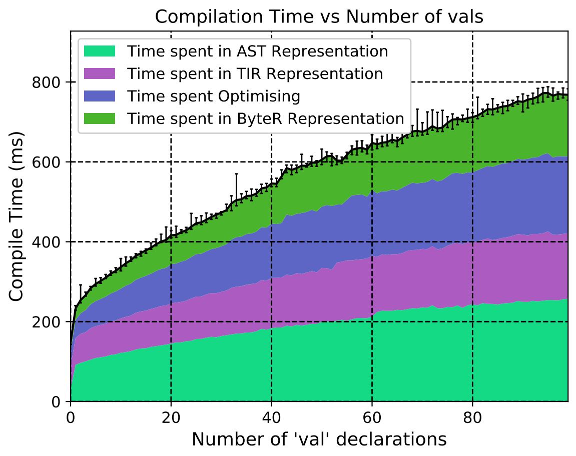


Figure 4.8: Comparing ML compilers in compilation time. The similarity to MLton reflects the similar goals. Both are whole-program optimisers intended to produce a single executable.

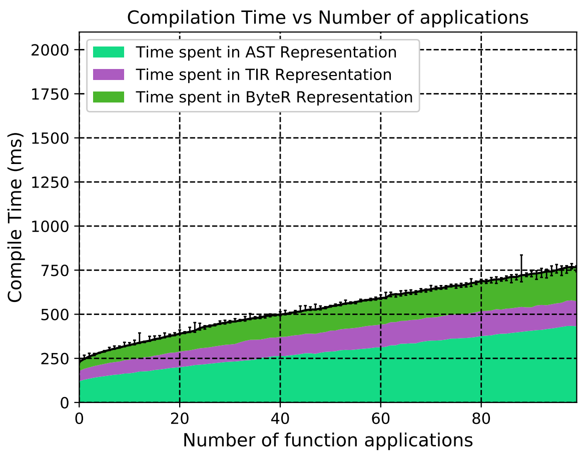


(a) Compile time without optimisations.

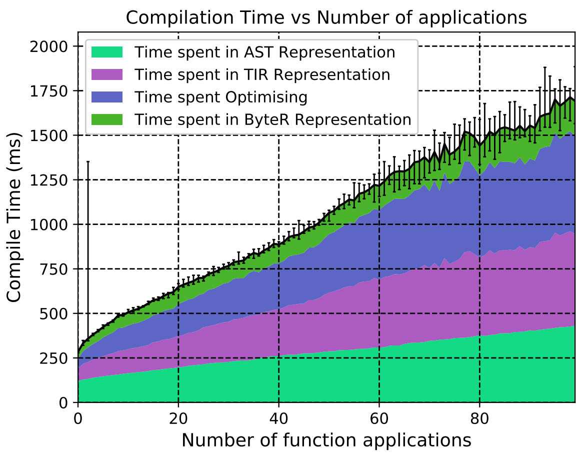


(b) Compile time with optimisations.

Figure 4.9: MLC's compile time against the number of `val` declarations.

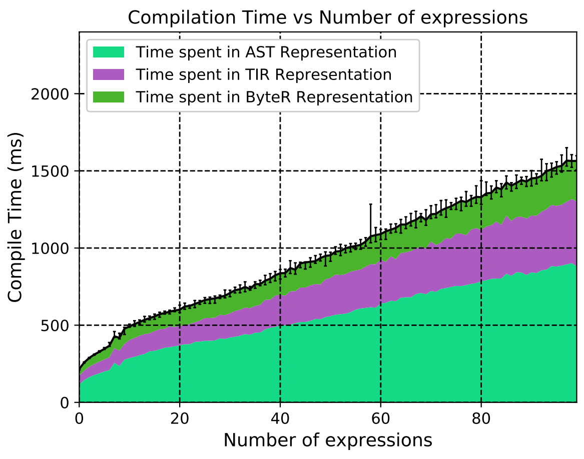


(a) Compile time without optimisations.

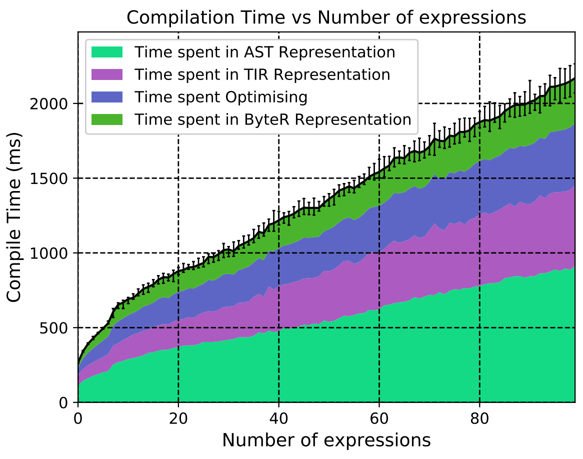


(b) Compile time with optimisations.

Figure 4.10: MLC's compile time against the number of function applications. Compile time increases drastically with optimisations due to inlining decisions. These result in the internal representation becoming around 300% bigger. This is found comparing printed TIR file length (with `--dump-inline`) excluding type environments and debugging messages. Most redundant internal structures are later eliminated, leaving a 5% executable size increase.

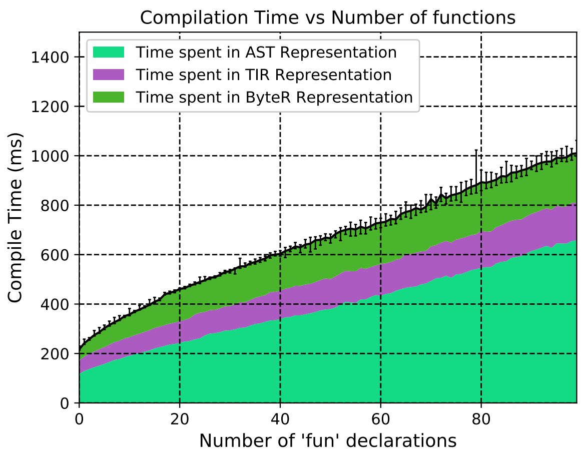


(a) Compile time without optimisations.

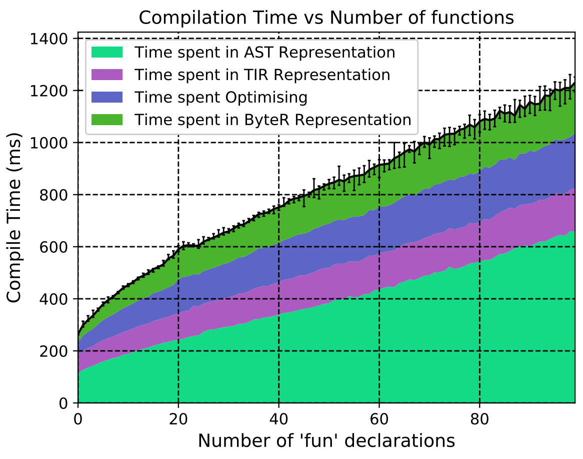


(b) Compile time with optimisations.

Figure 4.11: MLC's compile time against the number of expressions.

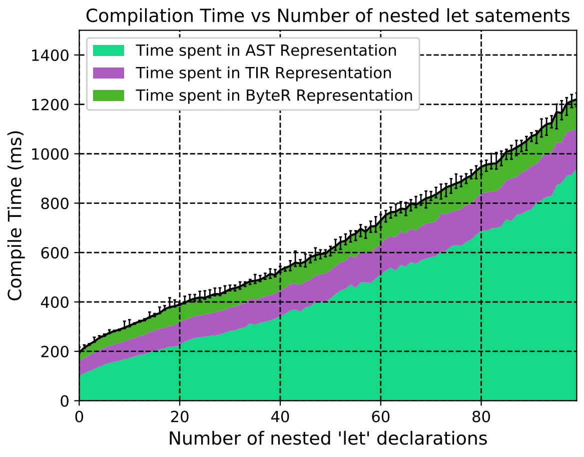


(a) Compile time without optimisations.

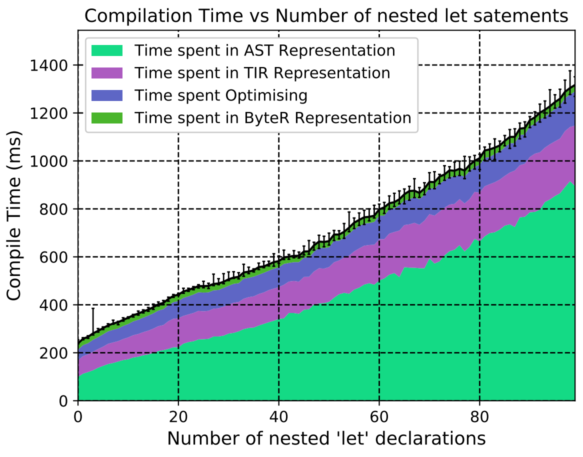


(b) Compile time with optimisations.

Figure 4.12: MLC's compile time against the number of functions.



(a) Compile time without optimisations.



(b) Compile time with optimisations.

Figure 4.13: MLC's compile time against number of nested `let` statements. This increases quickly due to a quadratic number of unifications required and the high overhead of each unification. In the optimised version, most code is removed by the simplify optimisation passes so little bytecode is produced.

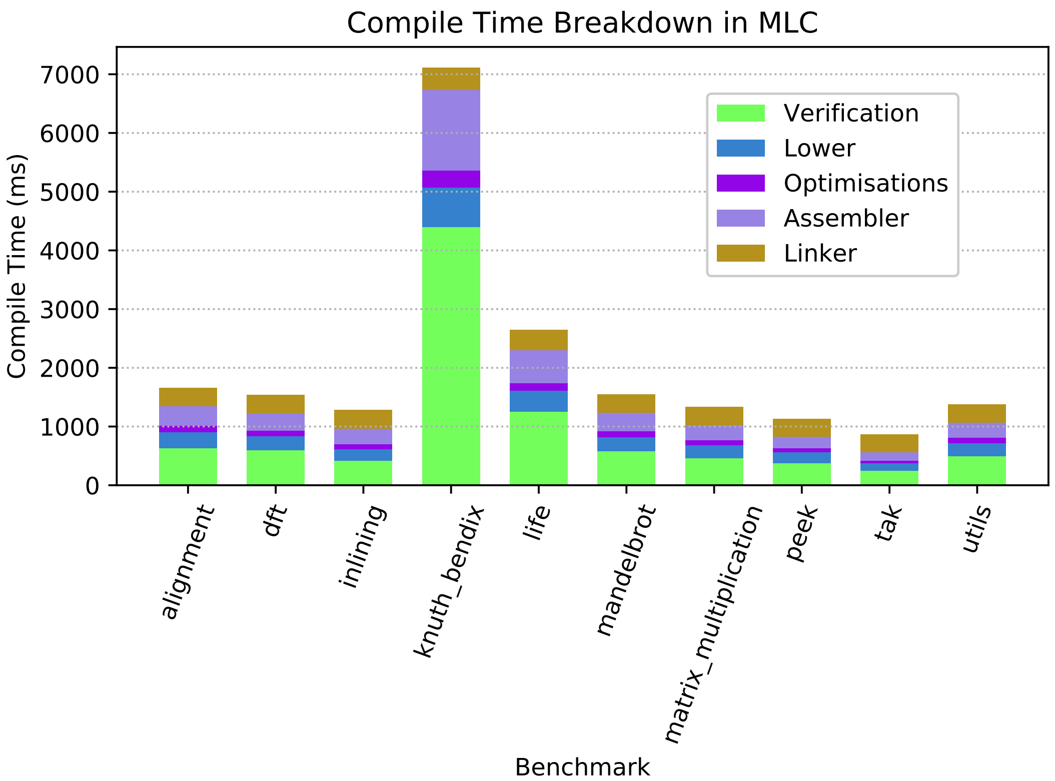


Figure 4.14: How MLC's compilation time is spent when compiling benchmarks.

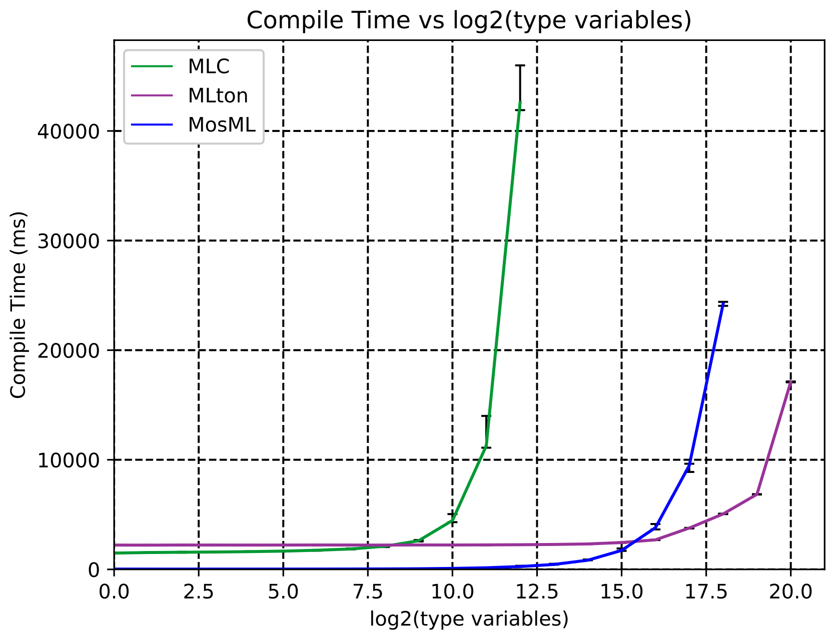


Figure 4.15: MLC's typechecker compared to MosML and MLton. Note scaling is the same, but happens sooner with MLC. This is because MLC stores a large number of unification variables past their lifetimes. This issue could be resolved by removing unneeded variables from unifiers, but would require bi-directional hash maps as discussed in Section 4.3.1 to do efficiently.

4.4 Code Size

Generated code size is an extremely important metric. Beyond distribution of code over low bandwidth links, which is (arguably) becoming less important, instruction caches are often not big enough for modern programs. A typical instruction cache is now 32 kB which can only hold 8192 fixed length 32 bit instructions.

This instruction cache limitation typically means code size is free until it isn't. For many embedded targets, executable size is significantly more important than executable speed. Traditionally, this is not the domain of functional languages but with suitable support that could change.

It is instructive to analyse effects of individual passes on code size. Figure 4.16 shows this. `simplify` has more effect on code size than on execution time time (cf. Figure 4.2) because many structures it eliminates are easy for the JIT to optimise away (for example the code in Figure 3.19) since they do not involve function calls. `tce` results in an increase in code size because the algorithm takes the function body and puts it inside a loop (see Section 3.3.1).

Finally, Figure 4.17 compares the size of generated executables between compilers. MLC performs well for two reasons. First, MLC benefits from the JVM's compact 8 bit instruction representation. MosML employs a similarly compact representation[3] to achieve its even better code size [42]. When run to produce a native executable, size regresses close to MLton. Note SML/NJ does not link the standard libraries, and requires the program to be loaded back into SML/NJ.

MLC's optimise-for-size option (`--optimize-size`) appears to have little effect over normal optimisation (`--optimize`) here. While this is true, the option has been introduced as it is unjustifiable to perform optimisations like inlining if small executable size is the aim.

[3]MosML benefits more than MLC as the MosML representation is designed for SML.

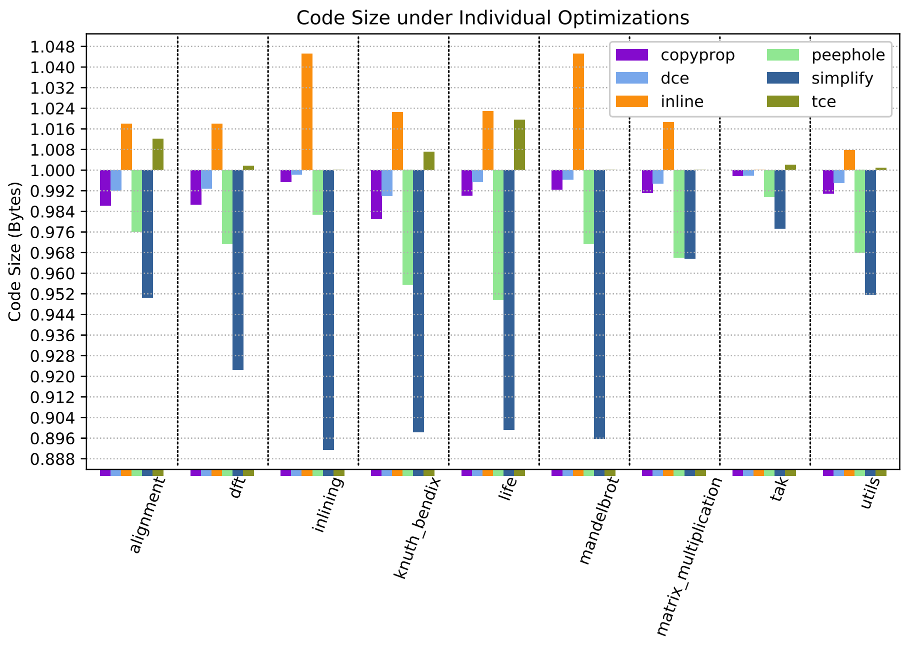


Figure 4.16: Code size under individual optimisations of generated executables.

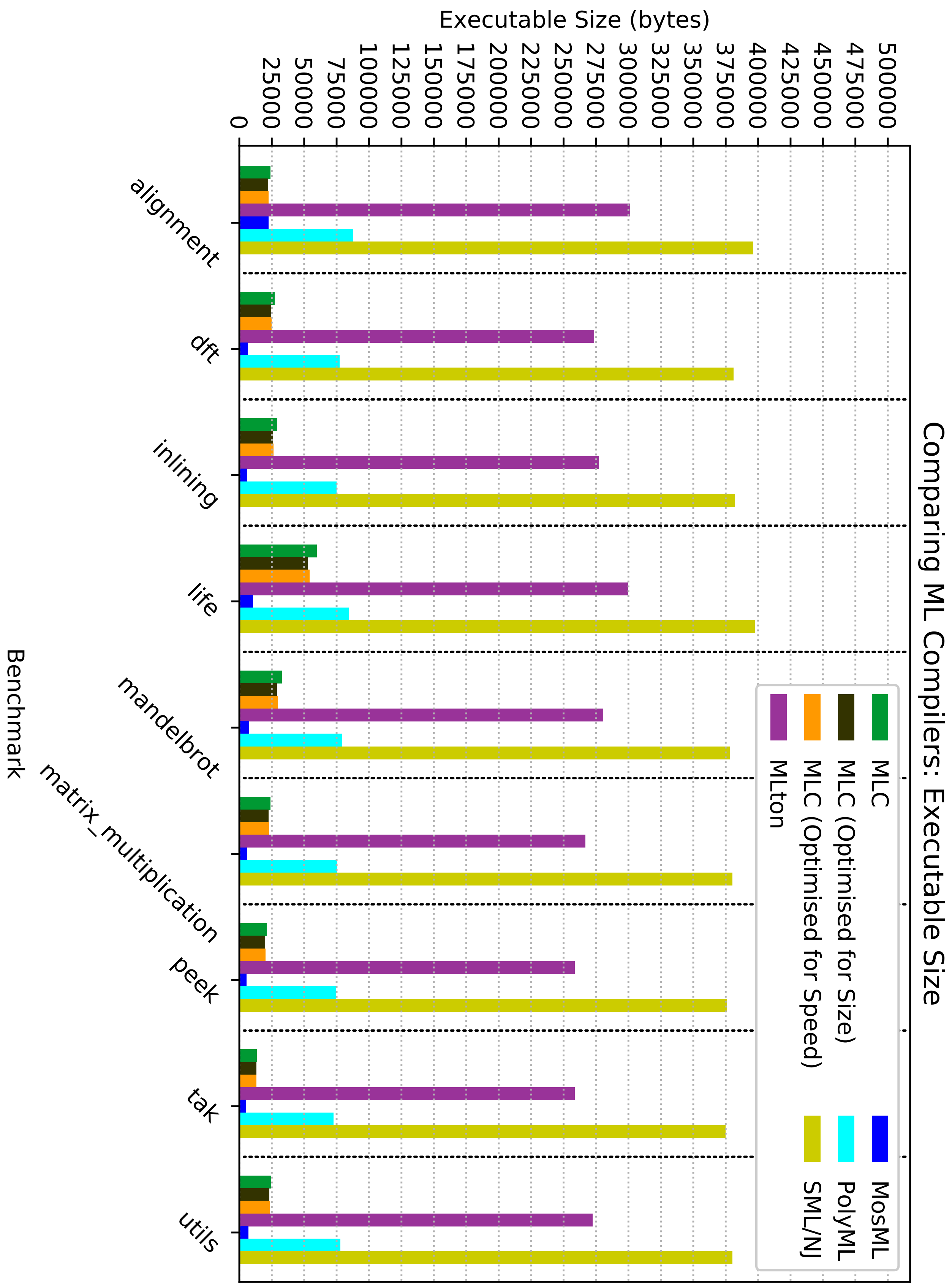


Figure 4.17: Comparing code size of generated executables for various ML compilers. There were small differences in source code to account for different entry points.

4.5 Additional Optimisations

Part of my evaluation evaluates generated assembly to determine which further optimisations are likely to be profitable. In addition to proposing several optimisations that have been mentioned previously in this dissertation, I propose the following optimisations would be useful:

Code Motion in the ByteR stage: The peephole pass removes most boxing. Some boxing and unboxing remains in `if` statements, e.g.

```
if (i == 0) {
  a = 1;
  b = new Integer(a);
} else {
  a = 0;
  b = new Integer(a);
}
c = b.intValue()
```

This could be removed with code motion, moving boxing out of the `if` statement, and using the peephole pass to clean up the adjacent box and unbox. I anticipate a 5% performance gain across my benchmarks.

Variable Merging: Variables never used during the same function call may share stack locations. Examples are variables inside `let` declarations on different branches of an `if` statement, or variables in different pattern matches. I expect similar impact to `copyprop`.

4.6 Summary

This section contains an analysis of the performance of MLC in three facets: execution time, compilation time and executable size. In all these facets, MLC has been shown to have acceptable performance.

We have seen the impact of each individual part of MLC. This pass-by-pass breakdown reveals many interesting effects that have the potential to influence further work. Data that provided a basis for decisions during the project have been presented. Much of the data presented here was collected in preliminary form throughout the project implementation. For example, the design of the `--optimize-size` flag was based entirely off data in Figure 4.16.

Chapter 5

Conclusion

My implementation of an optimising ML compiler works well. The optimisations produce noticeable effects on generated bytecode and fit nicely in the compilation pipeline, reducing complexity of tricky compilation stages.

The end product is usable and shrink-wrapped in a shell script to hide complex linking stages. The produced `jar` files are efficient and comparable in speed to current ML compilers.

While understanding the semantics of SML, I identified several issues with SML/NJ. Two of these were already reported, but I reported four additional issues [43, 44, 45, 46].

The iterative development approach I ultimately used was suitable for the task. However, there were cases where it was important to have more structure to my development (such as determining how SML language features should be represented on the JVM). In these cases, I found it better to approach the task with a waterfall methodology, preparing requirements documents as appropriate.

I was able to stick to my timetable almost to the day. I used my first slack week over Christmas early as I struggled to obtain remote access to my benchmarking machine.

5.1 Further Work

In retrospect, I omitted some optimisations that I subsequently considered to be important. For example, my proposal did not plan to implement copy propagation. As discussed in Section 2.3, I found it difficult to anticipate what MLC would generate suitable for removal (e.g. variable copies discussed in Section 3.1.7). I would have allocated an extension with an unspecified optimisation to be implemented if MLC had generated code that I had not expected, as was the case with copy propagation.

With an eye to the future, some time-efficient feature extensions to implement would allow a direct interface to Java standard libraries. I anticipate a casting based approach would not be difficult, although the libraries would be limited to returning primitives, strings or boxed primitives. Another easy extension with potential use would be allowing interfacing with Java code. This would require some simple changes to the name uniquifier. The JVM handles debugging information as part of the specification. Implementing this would be extremely useful and not particularly challenging. Parallelisation of generated code is another potential optimisation. This is a difficult task, but GHC has made some moderately successful approaches that provide a starting point [47]. Consideration of optimisations to aid the MLton benchmarks would be useful. Finally, implementing a larger language subset would be easy and aid expressiveness greatly.

5.2 Summary

I have demonstrated that pure functional languages can be compiled to the JVM. It would be interesting to know if MLC could easily be extended to provide a fully featured ML compiler. As the introduction outlines, I expect a fully featured SML-to-Java bytecode compiler would drive other SML compilers to provide easier interoperability with popular languages and other features provided by MLC. My compiler could provide easy integration with large Java projects, potentially providing good reason for companies to use SML in the real world. The framework I have provided should be easy to adapt to a new functional language, which can provide a base for functional languages to be implemented on the JVM much like LLVM provides a base for new imperative languages to be implemented.

Overall, this project has been a success. As specified in my success criteria, I implemented a compiler with multiple optimisations, and I have compared the effects of those optimisations in many different dimensions. I can see performance improvements in the region of 30%.[1] My project proposal originally asked, “Are optimisations helpful beyond existing JVM capabilities?” This project shows that the answer to this question is a resounding yes.

[1]This figure includes the peephole pass, so is not reflected in Figure 4.5. It is best seen in Figure 4.1.

Bibliography

[1] SML/NJ Project. Standard ML of New Jersey Home Page. `https://www.smlnj.org/smlnj.html`, 1996. Online; Accessed 2017-12-31.

[2] Michael Norrish. New year musing; workflow, and MoscowML pains. `https://sourceforge.net/p/hol/mailman/message/5862941/`, 2007. Online; Accessed 2018-01-12.

[3] Raja Vallée-Rai. SOOT: A JAVA BYTECODE OPTIMIZATION FRAMEWORK. 2000. School of Computer Science, McGill University, Montreal.

[4] John Hughes. *Research Topics in Functional Programming.* Addison-Wesley, 1990.

[5] École Polytechnique Fédérale de Lausanne. Introducing Scala. `http://www.scala-lang.org/old/node/25.html`, 2012. Online; Accessed 2018-03-12.

[6] Persimmon IT. The MLj Compiler Home Page. `http://www.dcs.ed.ac.uk/home/mlj/doc/index.html`. Online; Accessed 2017-12-31.

[7] LLVM Project. Expressive Diagnostics. `http://clang.llvm.org/diagnostics.html`, 2014. Online; Accessed 2018-03-12.

[8] Robin Milner, Mads Tofte, Robert Harper, David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[9] Oracle America. Limited License Grant. `https://docs.oracle.com/javase/specs/jvms/se9/html/spec-license.html"`, 2017. Online; Accessed 2018-03-29.

[10] Simon Hammond, David Lacey. Loop Transformations in the Ahead-of-Time optimization of Java Bytecode. *Compiler Construction*, 2006. Lecture Notes in Computer Science, vol 3923.

[11] *perf(1) perf Manual.* Accessed; 2018-02-01.

[12] John Henning. Benchmarks; good, bad, difficult and standard. `https://spec.org/cpu2017/Docs/overview.html#AboutBenchmarks`, 2017. Online; Accessed 2018-03-12.

[13] Jason Zaugg, P. Simon Tuffs. SBT One JAR. `https://github.com/sbt/sbt-onejar`, 2018. Online; Accessed 2018-03-18.

[14] Jackson Woodruff. MLC. `https://github.com/j-c-w/mlc`. Online; Accessed 2017-12-31.

[15] Travis CI. Travis MLC. `https://travis-ci.org/j-c-w/mlc`. Online; Accessed 2017-12-31.

[16] Scala Project. Scala Parser Combinators. `https://github.com/scala/scala-parser-combinators`. Online; Accessed 2017-12-31.

[17] Scallop. A Scala CLI Parsing Library. `https://github.com/scallop/scallop`. Online; Accessed 2017-12-31.

[18] Krakatau. Java Decompiler, Assembler and Disassembler. `https://github.com/Storyyeller/Krakatau`. Online; Accessed 2017-12-31.

[19] Andrew Cumming. Diversion: Mandelbrot sets. `http://www.soc.napier.ac.uk/course-notes/sml/mandel.htm`. Online; Accessed 2017-12-31.

[20] "LLVM Project". LNT – LLVM Performance Tracking Software. `http://llvm.org/docs/lnt/index.html`, 2018. Online; Accessed 2018-05-11.

[21] Thomas Williams, Colin Kelley and many others. *gnuplot(1) General Commands Manual.* Accessed; 2018-05-12.

[22] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[23] Oracle. jar – The Java Archive Tool. `https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jar.html`, 2017. Online; Accessed 2018-01-12.

[24] Andreas Rossberg. Standard ML Grammar. `https://people.mpi-sws.org/~rossberg/sml.html`. Online; Accessed 2017-12-31.

[25] Free Software Foundation. FAQs about GNU Licenses. `https://www.gnu.org/licenses/gpl-faq.en.html#GPLPlugins`. Online; Accessed 2017-12-31.

[26] Curtis Dunham, Matthew Fluet. MLton Performance. `http://mlton.org/Performance`, 2017. Online; Accessed 2018-01-12.

[27] Luis Damas, Robin Milner. Principal type-schemes for function programs. *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, 1982.

[28] Jessica Pacqette. Interprocedural MIR-level Outlining Pass. `http://lists.llvm.org/pipermail/llvm-dev/2016-August/104170.html`, 2016. Online; Accessed 2018-03-12.

[29] Luc Maranget. Compiling Pattern Matching to Good Decision Trees. `http://moscova.inria.fr/~maranget/papers/ml05e-maranget.pdf`, 2008. Online; Accessed 2017-12-31.

[30] Scott Owens, Michael Norrish, Ramaba Kumar, Magnus O. Mygreen, Yong Kiam Tan. Verifying Efficient Function Calls in CakeML. *Proc. ACM Program. Lang.*, 2017. Article 18.

[31] Matthew Fluet. SXMLSimplify. `http://mlton.org/SXMLSimplify`. Online; Accessed 2017-12-31.

[32] Manuel Lopez Ibanez et al. GCC Wiki Speedup Areas. `https://gcc.gnu.org/wiki/Speedup_areas`, 2015. Online; Accessed 2018-03-12.

[33] Timothy Jones. Optimising Compilers. `http://www.cl.cam.ac.uk/teaching/1718/OptComp/notes.pdf`, 2018. Online; Accessed 2018-03-18.

[34] Matthew Fluet, Henry Cejtin, Suresh Jagannathan, Stephen Weeks. MLton SSA Simplify. `https://github.com/MLton/mlton/blob/master/mlton/ssa/simplify.fun`, 2017. SML Source Code. Online; Accessed 2018-01-01. Commit a2b9f79.

[35] Robert Love. *taskset(1) User Commands Manual.* Accessed; 2018-05-11.

[36] T.F. Smith, M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

[37] Lawrence C. Paulson. *ML For the Working Programmer.* Cambridge University Press, New York, NY, USA, 2nd edition, 1996.

[38] Donald Knuth, Peter Bendix. Simple Word Problems in Universal Algebras'. 1983.

[39] Martin Gardner. Mathematical games. *Scientific American*, 223(4):120–123, 1970.

[40] J. McCarthy. An interesting lisp function. *Lisp Bull.*, (3):6–8, December 1979.

[41] Dennis Morse, Mike Frysinger, Roland McGrath, Paul Smith. *make(1) User Commands Manual.* Accessed; 2018-02-01.

[42] Sergei Romanenko, Claudio Russo, Peter Sestoft. Moscow ML Owner's Manual, 2000.

[43] Jackson Woodruff. Compiler crash when handling large reals. `http://smlnj-gforge.cs.uchicago.edu/tracker/index.php?func=detail&aid=191&group_id=33&atid=215`. Online; Accessed 2017-12-31.

[44] Jackson Woodruff. Missing warning for nonexhaustive valbind patterns. `http://smlnj-gforge.cs.uchicago.edu/tracker/index.php?func=detail&aid=188&group_id=33&atid=215`. Online; Accessed 2017-12-31.

[45] Jackson Woodruff. Unexpected exception in SML/NJ with invalid list pattern match. `http://smlnj-gforge.cs.uchicago.edu/tracker/index.php?func=detail&aid=190&group_id=33&atid=215`. Online; Accessed 2017-12-31. Closed as duplicate then reopened and accepted as a bug.

[46] Jackson Woodruff. Typechecker hangs with large number of curried arguments. `https://smlnj-gforge.cs.uchicago.edu/tracker/?func=detail&atid=215&aid=197&group_id=33`. Online; Accessed 2018-03-25.

[47] Tim Harris, Simon Marlow, Simon Peyton Jones. Haskell on a Shared-Memory Multiprocessor. *ACM SIGPLAN workshop on Haskell*, pages 49–61, 2005.

Appendices

Appendix A

Inlining Performance Profiling in Java

FunctionInterface.java

```java
public interface FunctionInterface {
  public Object apply(Object arg);
}
```

InlineExample.java

```java
import java.util.Random;

public class InlineExample {
  public static void main(String[] args) {
    // Fill an array with random data.  This stops the
        JVM from doing
    // value speculation.
    Random rand = new Random();
    int[] array = new int[100000000];
    for (int i = 0; i < array.length; i ++) {
      array[i] = rand.nextInt();
    }

    long startTime = System.currentTimeMillis();
    int sum1 = testNoInline(array);
    long noInlineEndTime = System.currentTimeMillis();
    int sum2 = testTraditionalFunCall(array);
    long traditionalEndTime = System.currentTimeMillis
       ();
    int sum3 = testInlined(array);
    long endTime = System.currentTimeMillis();
    // No Inline times
    System.out.print(Long.toString(noInlineEndTime -
       startTime) + ",");
    // Traditional Call times
    System.out.print(
        Long.toString(traditionalEndTime -
           noInlineEndTime) + ",");
    // Inlined Times
    System.out.print(Long.toString(endTime -
       traditionalEndTime));
  }

  public static int testNoInline(int[] array) {
    int sum = 0;
    for (int i = 0; i < array.length; i ++) {
      // This function creation pattern represents how
         functions
      // are to be represented in my compiler.
      Fuction fun = new Fuction(sum);
      sum = (Integer) fun.apply(array[i]);
    }

    return sum;
  }

  public static int testInlined(int[] array) {
    int sum = 0;
    for (int i = 0; i < array.length; i ++) {
      sum += (Integer) array[i] + (Integer) 1;
    }

    return sum;
  }

  public static int testTraditionalFunCall(int[] array
     ) {
    int sum = 0;
    for (int i = 0; i < array.length; i ++) {
      sum += sumFun(sum, array[i]);
    }

    return sum;
  }

  public static int sumFun(Integer i, Integer j) {
    return i + j;
  }
}
```

```java

class Fuction implements FunctionInterface {
  Object arg;

  public Fuction(Object arg) {
    this.arg = arg;
  }

  public Object apply(Object i) {
    return (Integer) i + (Integer) this.arg;
  }
}
```

Appendix B

Scala Garbage Collector

The JVM garbage collector (used by Scala) does not detect dead variables within functions. This is problematic for compilers with immutable tree representations that create instances of this tree many times but maintain references to old trees in a top level function. The following examples demonstrate this (consider `LargeObject` a program to be compiled).

AllocMany.java

```java
/* An example of the Java Garbage collector failing
 * for many large structures.
 *
 * Build as:
 *
 *  javac AllocMany.java
 *
 * run as:
 *
 *  java -Xmx5m AllocMany
 *
 * An exception is thrown attempting to allocate 'l4'
 * due to a lack of heap space (even though l1, l2
 * and l3 could all be collected, the JVM does not
 * notice this).
 */
public class AllocMany {
  public static void main(String[] args) {
    LargeObject l1 = new LargeObject();
    LargeObject l2 = new LargeObject();
    LargeObject l3 = new LargeObject();
    LargeObject l4 = new LargeObject();
  }
}

class LargeObject {
  int member[];
  public LargeObject() {
    member = new int[500000];
  }
}
```

AllocSingle.java

```java
/* Compile and run identically to AllocMany.java.
 *
 * This example does not crash due to lack of heap
 * space.
 */
public class AllocSingle {
  public static void main(String[] args) {
    LargeObject l1 = new LargeObject();
  }
}

class LargeObject {
  int member[];
  public LargeObject() {
    member = new int[500000];
  }
}
```

Computer Science Tripos – Part II – Project Proposal

An optimising compiler for ML

J. Woodruff, Magdalene College
Originator: J. Woodruff
May 11, 2018

Project Supervisor: Dr T Jones

Director of Studies: Dr J Fawcett

Project Overseers: Hatice Gunes & Robert Watson

Introduction

ML is not regarded as a particularly performant programming language. However, with refs stripped out, there is no reason it should be less amenable to optimisation than, say, Haskell, which is regularly regarded as a performant functional language.

This project aims to explore what could be done to make SML pereformant on the JVM. I intend to first develop a compiler for a subset of ML to Java bytecode and then write optimisations for this compiler. To goal is to analyse why performance improves (or does not improve), and to find possible sources of additional performance.

Java bytecode is again not highly regarded for its performance characteristics (particularly when one disables JIT for more understandable benchmark results). I have chosen the JVM as a target due to its simplicity, but a central idea underlying this project is that compiling to the JVM is not significantly conceptually different from compiling to native code.

The rest of this document details my ideas and how they are to be achieved.

Starting point

I intend to write the project in Scala. I already have some experience in Scala. (I wrote my A-Level project in it.) Further, there is a parser-generator library for Scala for which I have verified the suitability for use. I also intend to build on what is already provided in the Scala standard library (for file IO etc).

My project will also involve some Python for testsuite scripts and benchmarking. I have some experience with Python. (Again, it was the language taught for my A-Level course).

I have looked through the Optimizing Compilers (Part II) course.

My compiler will compile to JVM bytecode. A project by the name of 'Krakatau' will be used for the assembler. This will not constitute part of my project. It will be treated as a separate tool. Again, I have verified that this assembler is suitable for the task.

Part of my evaluation is to be done with LNT (LLVM Nightly Testing), which is a project that offers performance tracking over time. I am familiar with the idea of the LNT project from my summer internship.

Lastly, I will use Git and sbt (simple build tool) to manage the complexity of the project.

Resources required

Development for this project will be done on my own laptop that runs Ubuntu 17.04. It has 8 GB of RAM and a quad-core Intel i5-5200U CPU running at 2.20GHz. Backup will be done using a GitHub repository. Should my laptop fail, I will use the MCS machines (installing the required development tools locally) until I can source a replacement.

I require non-shared access to a machine to do the benchmarking. I require a lab account for this, and Dr Jones has agreed to sponsor this.

Work to be done

The project breaks down into the following sub-projects:

1. The construction of a working ML compiler for the subset defined below.

   This is broken into:

   (a) A parser, provided by a parser generator in Scala. I will write the description of the grammar for this.

   The abstract syntax tree generated will be structure for structure identical to SML.

   (b) A type checking pass. I will implement the Hindley-Milner type checking algorithm as is standard for ML.

   (c) A conversion pass into an intermediate format suitable for analysis. This language will be similar to the original abstract syntax tree.

   (d) A lowering pass to convert this into a format close to JVM assembly.

   A key parts of this will be to convert functional representations into object oriented representations.

   Roughly, each JVM instruction will correspond to one node in the (flat) tree at this point.

2. Construction of essential testing infrastructure. Namely, a test suite and an infrastructure for benchmarking.

3. Writing relevant and useful benchmarks. In early Michaelmas, I intend to write a small number of micro benchmarks that solve relevant, not completely trivial problems amenable to being benchmarks.

   These benchmarks are to be constructed in a manner that reflects my optimisations.

4. A range of optimisations. These are detailed in the plan.

Since the implementation is in Scala, my design will feature object-oriented design principles. Notably, this means that the various intermediate representations will be class hierarchies rather than data types (such as in Tim Griffin's compilers course).

The subset of ML I intend to compile includes functions and higher-order functions. I will drop other language features to make time for optimisations.

Standard libraries will be provided as needed for the benchmarks.

For the sake of variety of experience and to make my project closer to a general optimizing compiler rather than a compiler that only optimizes a particular case, I have selected multiple optimisations rather than focusing on one.

Methodology

I will use a waterfall methodology with feedback.

The waterfall methodology suits this project because it can be compartmentalised well, with each pass being treated as its own small waterfall.

I intend to use the feedback for situations in which subsequently implemented passes demand features that were not originally considered (obviously I want to keep this to a minimum, but I do believe that such situations will arise).

Success criteria

I will consider this project a success if I successfully implement a compiler that performs some optimisations. I will evaluate these by either examining why they increase performance or why they are not behaving as expected. This performance analysis will be done using multiple microbenchmarks. Microbenchmarks used will be designed to solve actual problems. Some will be written in a style designed to showcase the optimizations I have written and others will be written in a more generic style. I will make some use of very constructed benchmarks (not solving a problem, but only demonstrating an optimization) to either demonstrate the maximum potential of each optimization or to really drill down on why things aren't performing better.

Compilation time is not a central aim of the project, but it is relevant to the extent that the compiler must be useable.

My core deliverables will therefore be a working compiler with at least one optimisation that I can analyse. I fully intend to implement more optimisations unless things go badly wrong.

Risks and mitigation

The biggest risk is that the compiler itself doesn't work and that I spend more time on that than planned. In this case, I consider that a peephole pass will be part of my core deliverables (This is chosen as I believe it will yield the most interesting analysis of why or why not performance has increased.)

Another risk is that the libraries I am relying on fail. If the parser-generator does not work, I will push the entire project back to re-implement the first work package in a different parser-generator. If the assembler fails, there is another (with a not identical, but similar structure) that I will try. If this also fails, then I will have to make drastic changes to allow time to do the assembly myself (possibly as modifications to Krakatau) with significant project delays. Should LNT fail, I will create the relevant graphs by hand (I will still have collected the benchmarking data).

Possible extensions

These extensions were selected as further optimisation passes that I believe will result in performance gains. They were also selected as optimisations I am interested in implementing. They are listed in order of preference. However, I may change this if I notice particular compiler behaviours that make some optimisations more profitable than others (such as if I notice my compiler is generating a lot of dead code, then implementing the dead code elimination pass will be ranked higher up).

1. Function specialization. Haskell specialization passes involve partially evaluating functions with case statements at compile time.

   The same technique can be used on the JVM, which has instructions specifically for the Java primitive types, to convert generic functions into specialised versions of functions that do not require method calls for every operation.

   I estimate that this will take about a week. It may or may not be applicable depending on exactly how my lowering pass is written.

2. Simplification of maths. Like the peephole pass, this is focused on creating a framework and adding a few examples. (e.g. transforming divisions by 2^n into shifts.)

   I expect this to take about a week.

3. Dead code elimination.

   I expect this to take a week. An implementation of DCE will be intended to clean up after other passes of my compiler rather than as a general DCE pass. Because it will only be expected to work in special cases, the time allocated to implement this is less than I would allocate for a full DCE pass.

4. Constant propagation.

   I expect this to take two weeks.

5. Partial inlining of recursive functions. This is an optimisation in which recursive functions are inlined once. This helps future passes work better.

I expect this to take a week.

6. Common subexpression elimination.

   I expect this to take two weeks.

Timetable

1. **04 Oct – 17 Oct** Read about the JVM, Krakatau formatting and SML. Write the auxiliary parts of the system (test suite scripts and benchmarking scripts).

   These tasks were chosen for these weeks because they are roughly independent of this document.

   The main deliverable here is this project proposal.

2. **18 Oct – 31 Oct** Begin the compiler part of the project. I expect to have the frontend finished, and the type checking pass completed.

   The deliverable during this slot is a working frontend type checking pass. This is to be verified by inspecting dump files from my compiler.

3. **1 Nov – 14 Nov** During these weeks my lecture load begins to decrease.

   I expect to implement the first lowering pass, converting between my SML-like IR and my optimisation IR.

   I also expect to plan out in detail how the second lowering pass will work. This will involve looking into the JVM assembler format in detail.

   There are two deliverables for this work package. I will verify that my first lowering pass works as intended. I also intend to create a document outlining my ideas on the mapping between my IR and the Java bytecode. The purpose of this document is to get advice from Dr Jones on things to be wary of in the next work package. It also exists as a design guide for the work package the following weeks.

4. **15 Nov – 28 Nov** During these weeks I intend to begin the construction of the last phase of the compilation. This will involve the lowering from my intermediate representation into a representation suitable for direct output to JVM bytecode.

   By the end of this work package, I expect to have compilation working for a subset of my chosen language subset. Namely, I expect to have simple functions (with only one case), case statements compiling and have set up the standard libraries I need.

5. **29 Nov – 12 Dec** In these weeks my lecture load will have decreased. The target during these two weeks is to finish the construction of the compiler. Namely, I will write the lowering for higher order functions and functions with cases.

   This work package also involves the output of JVM assembly, but the last IR is to be designed so that this task is trivial.

   The deliverable for these two weeks is to have a working, end-to-end compiler.

6. **13 Dec – 26 Dec** First optimisation passes will be written. In particular, I expect to implement:

- A peephole pass. Peepholes will continue to be added throughout the project, as deemed fit by the generated assembly.
- Function inlining.

  This is the optimisation I expect to see the most performance gain based on limited tests with Java

The deliverable for these weeks is to have both of these optimisations working. This will be verified by inspecting code generation.

I expect that my primary goals will be met by this stage.

7. **27 Dec – 9 Jan** I expect to implement:

   - Elimination of tail recursion.

   The concrete verification that these are working is to be done again by inspecting code generation. These weeks are intentionally left light as break weeks and slack weeks if needed.

8. **10 Jan – 23 Jan** In these weeks I would like to implement one or more of my optional extensions (whether it is one or more depends on which I select).

   I will take some time in these weeks to write the progress report.

   The deliverable is to see that the optimizations have been implemented (by looking at generated code).

9. **24 Jan – 6 Feb** In these weeks, I intend to implement another of my extensions.

   These are also allocated as slack weeks should things from previous weeks slip or should I decide that I need more data than I had previously managed to collect.

   The deliverable here is to see that an extension has been implemented. (again to be seen from the generated code).

10. **7 Feb – 20 Feb** Gathering additional benchmarking data. This will involve writing benchmarks in addition to what was previously written. These benchmarks will again be designed to have real world use cases, but will not be designed to demonstrate the optimisations of my compiler.

    I will also write and rehearse the presentation early in this work package.

    I expect to be able to see elimination of tail recursion working (again, by inspecting code generation) and have some benchmarking data for use in future weeks. These are my deliverables for this period.

11. **21 Feb – 6 Mar** The main deliverable here is to have a working draft of the final dissertation.

12. **7 Mar – 20 Mar** Iteration over dissertation (and revision).

13. **21 Mar – 3 Apr** Continued iteration over dissertation (and revision).

14. **4 Apr – 17 Apr** Mainly focusing on revision at this point, incorporating any feedback received.

15. **18 Apr – 2 May** Proof reading and then an early submission so as to concentrate on examination revision.